NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A FRAMEWORK FOR MAXIMIZING THE SURVIVABILITY OF NETWORK DEPENDENT SERVICES

by

Baris Aktop

March 2003

Thesis Advisor:	Geoffrey Xie
Second Reader:	John Gibson

Approved for public release: distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: A Framework for Maximizing the Survivability of Network Dependent Services			**5. FUNDING NUMBERS**
6. AUTHOR(S) Baris AKTOP			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release: distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

As a consequence of the developments in information technology and the Internet, the world is getting increasingly dependent upon distributed systems and network services. Unfortunately, the security of these services has not kept pace with the advances in information technology itself. Security practitioners accept that, a system that is connected to an unbounded network, e.g., the Internet, will be vulnerable to attacks regardless of its security features. However, the emerging discipline of survivability can help ensure that such systems deliver essential services and maintain essential properties, such as integrity, confidentiality and performance, despite the presence of intrusions.

Although survivability has been accepted as a means of sufficiently addressing the security problems of current network services, unfortunately, the studies that have been done on network survivability so far are not mature enough and they lack quantifiable metrics.

To address this lack of network survivability measure, a global connectivity metric is developed in this thesis. Additionally, an election protocol based on this metric is designed for the SAAM prototype to enhance the survivability of the SAAM server.

14. SUBJECT TERMS Network survivability, fault tolerance, SAAM, maximizing network service survivability			**15. NUMBER OF PAGES** 147
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release: distribution is unlimited

A FRAMEWORK FOR MAXIMIZING THE SURVIVABILITY OF NETWORK DEPENDENT SERVICES

Baris AKTOP
Lieutenant Junior Grade, Turkish Navy
EE, Turkish Naval Academy, 1997

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2003

Author: Baris AKTOP

Approved by: Geoffrey Xie, PhD
Thesis Advisor

John Gibson
Second Reader

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

As a consequence of the developments in information technology and the Internet, the world is getting increasingly dependent upon distributed systems and network services. Unfortunately, the security of these services has not kept pace with the advances in information technology itself. Security practitioners accept that, a system that is connected to an unbounded network, e.g., the Internet, will be vulnerable to attacks regardless of its security features. However, the emerging discipline of survivability can help ensure that such systems deliver essential services and maintain essential properties, such as integrity, confidentiality and performance, despite the presence of intrusions.

Although survivability has been accepted as a means of sufficiently addressing the security problems of current network services, unfortunately, the studies that have been done on network survivability so far are not mature enough and they lack quantifiable metrics.

To address this lack of network survivability measure, a global connectivity metric is developed in this thesis. Additionally, an election protocol based on this metric is designed for the SAAM prototype to enhance the survivability of the SAAM server.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to acknowledge the great Turkish nation and the Turkish Naval Forces for providing me this wonderful education opportunity.

I would first like to thank my parents, Ahmet and Guler Aktop for the love and guidance they gave me throughout my life. I would not be where and what I am today without their unwavering support and sacrifice.

I would most like to thank my wife, Sebnem, and my daughter, Ozge for their support, love, and understanding throughout my thesis work. Without their unselfish love and support, I would have never been able to complete my thesis.

I would also like to thank and extend my most sincere gratitude to Professor Geoffrey Xie and Mr. John Gibson for their great help with my thesis work. Without their guidance and editorial comments, this thesis work would have not been completed.

Finally, I would like to dedicate this thesis work to my beloved daughter Ozge Aktop.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION AND OVERVIEW

A. BACKGROUND

As a consequence of developments in information technology and the Internet, the world is getting increasingly dependent upon distributed systems and network services. Unfortunately, the security of these services has not kept pace with the advances in information technology itself. It has been accepted by most of the security practitioners that "no amount of system hardening can assure that a system that is connected to an unbounded network, e.g., the Internet, will be invulnerable to attack" [KYA00]. However the discipline of survivability can help ensure that such systems will deliver essential services and maintain essential properties, such as integrity, confidentiality and performance, despite the presence of intrusions. As an emerging discipline, survivability builds on related fields of study, which include security, fault tolerance, safety, reliability, performance, verification, and testing [ELL99].

Although survivability has been accepted as a means of sufficiently addressing the security problems of current network services, no study or effort for quantifying survivability of network services has been found during the literature research period of this thesis work. Most of the recent papers on survivability are written from the perspective of software development and design process. The implementation of these survivability measures usually involve a considerably large amount of money and time since they require a rigorous engineering process and, sometimes, big changes to existing protocols or systems. However existing network services can be made more survivable by focusing on fault tolerance and reliability factors affecting survivability.

To enhance the fault tolerance properties of the Server Agent-based Active network Management (SAAM) system, Scott Margulis, who was a graduate student at Naval Post Graduate School (NPS), took a major step. He added the Margulis AGent-based Mobile Applications (MAGMA) functionality to SAAM, allowing the server to be transferred to another router node in a SAAM region with the help of software agents. Details on his work can be found in his thesis. [MAR01]

Margulis achieved his goal in moving the server around the network but he did not address the decision problems of "when" and "where" to move the SAAM server. To address these problems a case study, in the form of a class project, was done by Baris Aktop, Ekrem Serin, and Selcuk Ozturk [AKT02]. In this study, an ad hoc model was developed to move the server around the SAAM region randomly and, yet, intelligently, based on a metric called "Fit-to-Serve-Index (FSI)". Although somewhat ad hoc, this model can be taken as a foundation for more rigorous studies to make network services more survivable.

B. APPROACH

Under its current configuration, the SAAM server already has a fault tolerance mechanism, which was designed by Scott Margulis in December 2001. This mechanism, MAGMA, considerably improved the fault tolerance posture of SAAM by allowing the server to be moved around the SAAM region. The idea behind Margulis's work was that the SAAM server at a fixed location would eventually be located by an attacker and would be intruded regardless of how complicated and strong its security measures were. It would be desirable that the SAAM server functionality is able to relocate to another router node before the attacker locates the server or makes it inoperable.

With this idea in mind, Margulis managed to move the server around the SAAM region using software agents installed on each and every router node in the SAAM region. Although the major part of the work was completed, the solution to the problem of when and where to move the server was left as future work. Therefore, an election protocol should be developed to make intelligent selections between candidate router nodes.

The proposed model in [AKT02] was taken as the foundation and the FSI metric was modified to include a global connectivity factor. The factor, as explained in Chapter IV, is the focus of this thesis work.

C. SCOPE

In the current SAAM prototype the server can be moved to another router node whenever necessary, but it has to be moved manually. This restriction severely affects the practicality of MAGMA. Therefore this process should be automated and human intervention must be taken out of the loop. In order to achieve this, an election protocol was developed. This election protocol used the FSI metric as explained in [AKT02]. The metric was modified to include the global connectivity factor. An open source Java-based Graph Algorithms Platform (JGAP) developed by Chen D. and made publicly available at http://jgap.sourceforge.net (20 November 2002) was modified and used to compute the connectivity factor.

Additionally, a Java program was written to identify all min-cuts between two nodes of a given graph. This program contributed to the thesis work of Eng Hong Chua, another graduate student working on the development of a heuristic for comparing the connection reliability of two nodes to a common destination node when these two nodes have the same number of edge-disjoint paths to that destination. The heuristic is based on estimating the probability of each of the nodes being cut from the server given same number of link failures. The Find All Mincuts program developed by this thesis allowed Chua to compare the results of his heuristic with the actual cut probabilities and test the accuracy of the heuristic.

THIS PAGE INTENTIONALLY LEFT BLANK

II. SURVIVABILITY IN COMPUTER NETWORK SERVICES

A. THE NEED OF SURVIVABILITY IN NETWORK SERVICES

Society is growing increasingly dependent upon large-scale, highly distributed systems that operate in unbounded network environments, which like the Internet, have no central administrative control and no unified security policy. Today, most of the sectors, such as energy, transportation, telecommunications, financial services, health care, and education, depend on networks working at different scales but mostly on a large scale. These large-scale networked systems have improved the efficiency and effectiveness of many organizations and presented them with new levels of integration on unbounded networks like the Internet. For instance, e-commerce which has generated major revenues for business requires an integration of systems of suppliers, customers, and financial organizations on the Internet.

However, this progress came with the price of elevated risks of system intrusion and compromise. Unfortunately, traditional security approaches are not sufficient to protect these systems. Since no individual component of a system can be made immune to all attacks, accidents, and design errors, augmenting traditional security approaches with survivability techniques is necessary. The discipline of network survivability and security helps to ensure that such systems can deliver essential services and maintain essential properties, such as integrity, confidentiality and performance despite the presence of intrusions [ELL99].

1. The Definition of Survivability?

In this thesis, survivability is defined as the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents [FIS99]. In this definition, the term mission refers to high-level organizational objectives, and the term mission fulfillment can be judged in the context of conditions that affect the achievement of mission objectives.

For example, a financial system may be down for 12 hours during a period of power outages caused by a hurricane. If the system preserves integrity and confidentiality

of data and resumes essential services following the period of downtime, it can reasonably be considered to have fulfilled its mission. However, if the system is down unexpectedly for 12 hours under normal conditions or minor environmental stress depriving users of essential financial services, it can be considered to have failed its mission, even if its integrity and confidentiality are preserved [LIN99].

Timeliness is also a very critical quality that attributes to a survivable system. If the system enters into a state where it cannot complete its mission within acceptable time limits then survivability is no longer a consideration.

Even though the terms attack, failure, and accident, which are used in the definition of survivability, have distinct meanings, they can all be classified as potentially damaging events that need not be identified or handled individually. This is due to the fact that a distinction between an attack, failure, or accident is less important than the impact of the event when looking at the problem from the survivability perspective. Survivability focuses on the effects of damaging events but not the source of them. The identification of cause can be made afterwards. Another important feature of survivability, which is usually misunderstood, is that it is the mission fulfillment that must survive not any particular subsystem or system component [LIN99]. Therefore identifying essential services and the essential properties supporting them is a very critical step in developing system survivability.

Survivable systems generally exhibit four key properties, given in Table 1, which determine their capabilities to deliver essential services.

Key Property	Description	Example
Resistance to attacks	strategies for repelling attacks	user authentication stochastic diversity of programs
Recognition of attacks and assessing the extent of damage	strategies for detecting attacks(including intrusions) and understanding the current state of the system, including evaluating the extent of damage	recognition of intrusion usage patterns internal integrity checking
Recovery of full and essential services after attack	strategies for restoring compromised information or functionality, limiting the extent of damage, maintaining or, if necessary, restoring essential services within the time constraints of the mission, restoring full service as conditions permit	replication and re-initialization of data
Adaptation and evolution to reduce effectiveness of future attacks	strategies for improving system survivability based on knowledge gained from intrusions	incorporation of new patterns for intrusion recognition

Table 1. The Key Properties of Survivable Systems [ELL97]

Survivability covers a broad range of engineering areas such as security and fault tolerance.

2. Survivability and Security

Traditional security approaches strongly focus on prevention, not on recovering or maintaining the services attacked. Information systems are always in either one of two states, secure or compromised, according to the traditional security approach. On the other hand, survivability finds this approach incomplete and it eliminates system hardening as sufficient for protecting information systems from successful attacks. Therefore, a survivability approach tries to maintain essential services of information systems even under or after attack, while retaining the preventive measures of traditional security approaches.

In summary, traditional security work contributes to system resistance to attacks but not to system robustness under attack, whereas survivability considers robustness under attack at least as important as system hardening or resistance to attacks. This distinguishes survivability from security.

3. Survivability and Fault Tolerance

Fault tolerance is another area of research related to survivability. It deals with the statistical probability of a failure or combination of failures due to accidents but not malicious attacks. For example, a web server connected to the rest of the network with three distinct links can only be separated from the network by disabling all three links, assuming that the other links of the network are reliable. Fault tolerance treats these link failure events as independent and computes the probability of separation accordingly. Thus, the probability of these three independent failures occurring simultaneously will likely be small. A typical fault tolerant design of the web server will probably not address this failure scenario because the probability of its occurrence is below some threshold value. However, an intelligent and motivated attacker can develop successful, orchestrated attacks to cause simultaneous failures to happen, separating the web server from the rest of the network. Therefore, to be considered survivable, the web server must have contingency plans to fill that gap.

Another factor that contributes to system survivability is redundancy, such as a back up for a web server, or an alternative path between two hosts. The problem, however, with redundancy is that it replicates the same vulnerabilities if homogenous components are used to create redundancy. Therefore, survivable systems should have multiple heterogeneous components offering the same services, thus preventing the failure of all components at the same time with a single type of attack.

a. Survivability Through Heterogeneity

Although not yet accepted widely, heterogeneity is believed to be another key factor contributing to survivability. Today's networking technology tends to converge on a single protocol or software application to achieve high interoperability

with reduced costs. On the other hand, this homogeneity makes each and every component of the network vulnerable to the same kinds of threats. For example, Microsoft products are widely used by a large number of companies and individuals. If an exploit is found in one Microsoft product, all users of this product will be potentially vulnerable. An example is the attack on e-mail applications by the "love bug". The attack exploits vulnerabilities in Microsoft Outlook but has no effect on UNIX e-mail clients. Thus, when deployed strategically, heterogeneity creates networks that are more resilient to orchestrated attacks, resulting in more survivable systems.

A network with homogeneous components invalidates the assumption, often used in design of fault tolerant systems, that the probability of each component failure is independent. In other words, for fault tolerant systems the probability of failure is computed assuming that component failures are independent. This assumption cannot hold when considering network based attacks in which multiple components can be victims. This is because once a successful attack is performed on one component; the same attack can also be performed on another identical component, resulting in failures that are not truly independent.

In conclusion, heterogeneity and diversity in implementations of components results in networks that are more resilient to attacks and more survivable.

B. CURRENT STATE OF THE ART IN NETWORK SURVIVABILITY

Much of today's research and practice in computer system survivability takes a security-based view of defense against computer attacks. Security usually applies to bounded networks where a central administrative security policy can be enforced. However, survivability covers more than just security. Survivability shares goals with other areas of the system and software quality including reliability, safety, dependability, modifiability, and fault tolerance [FIS99]. Furthermore, the survivability approach can be applied to unbounded networks like the Internet. Advances in areas of software quality often contribute to survivability. For example, while increased security can improve the general capacity of a system to resist attacks, improvement in the reliability or quality of the code can eliminate bugs that could be exploited by an intelligent adversary. Ease of

modification can allow vulnerabilities to be quickly repaired, and can support the evolution of a system to maintain survivability over time [MEA99].

Most of the current practices of survivability are related to software development processes. One such process, the Survivable Network Analysis method, is explained below to provide more insight into the latest survivability studies.

Survivability is a recent concept and, unfortunately, the current practice of survivability is not mature enough. Most of the studies done on survivability usually stay on the level of defining survivability and do not attempt to quantify survivability. As stated in [XIE02] Ellison provided a general definition of network survivability in [ELL99] and described some solution approaches to the problem. A follow-on paper by the same authors defined a software engineering process for designing survivability into application [MEA00]. Some more concrete results are presented in some contemporary papers, e.g., [SUL99], [UMA01], [WEL00]. However, their focus was still on how to make software agile to effectively detect and react to system component failures and software errors. Interestingly, in [WEL00], a customizable utility function is used to indirectly measure survivability of a system configuration from the point of view of the system user. In [JHA00] Jha and Wing proposed a formal framework based on Bayesian networks for reasoning about the survivability properties of distributed systems. The work was rigorous but the proposed algorithm was too complex for large networks.

In summary, early studies have shown survivability as an important aspect of today's computer network systems. But the state of art of survivability practices still fixates a security point of view. There is no genuine theoretical framework of survivability. Much work has to be done in this area. There is no doubt survivability will be one of the most interesting and challenging areas of research in computer networks

1. Survivable Network Analysis

Survivable Network Analysis (SNA) was developed by the SEI Computer Emergency Response Team (CERT) Coordination Center at Carnegie Mellon University as a practical engineering process for systematic assessment of proposed systems, existing systems, and modification to existing systems. The SNA method, depicted in

Figure 1, is purposed to help organizations understand survivability in the context of their operating environment [LIN99].

key survivability properties of resistance, recognition, and recovery.

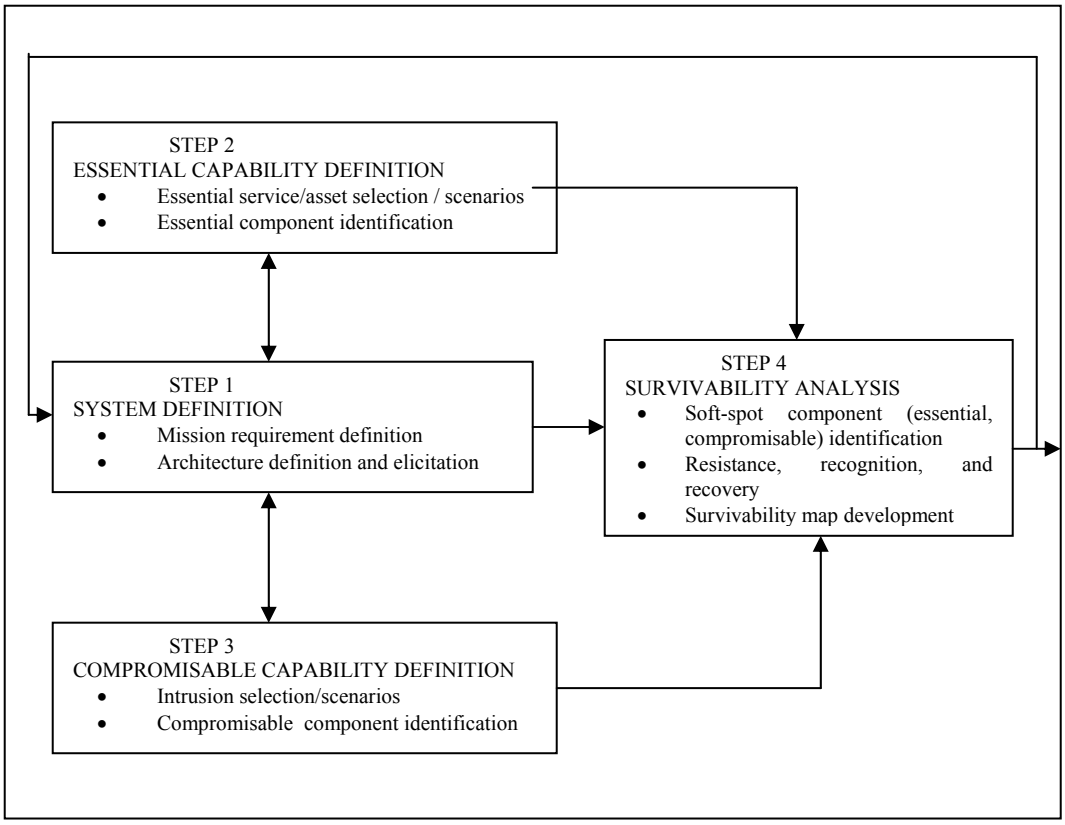

Figure 1. The Survivable Network Analysis Method [LIN99]

a. Step 1: System Definition

First, mission objectives, system requirements, structure and properties of system architecture, and risks in operational environments are identified.

b. Step 2: Essential Capability Definition

Second, essential services that should survive and essential assets that should maintain its properties such as integrity are identified.

c. Step 3: Essential Capability Definition

Third, compromisable components are identified based on the scenarios developed related to environmental risks.

d. Step 4: Survivability Analysis

Finally, soft spot components, which are both essential and compromisable, are identified. Then these components are analyzed on the basis of the key survivability properties of resistance, recognition, and recovery.

III. SAAM AND ITS FAULT SURVIVABILITY POSTURE

A. SAAM ARCHITECTURE AT A GLANCE AND ITS HISTORY

1. How SAAM Originated and How It Works

The conventional network architecture is based on stand-alone routers, and the processing capacity and memory of these routers may be overloaded with the introduction of integrated services and other network traffic engineering requirements. To address this problem, the Server Agent-based Active network Management (SAAM) approach was proposed by Geoffrey Xie *et al* in 1998[XIE98] SAAM is based on a client-server like hierarchical model where a server makes routing decisions for a number of routers. The SAAM architecture relieves individual routers of complicated routing and network management tasks required for providing quality of service (QoS). Since the SAAM servers make the routing decisions, the routers become "light-weight" with respect to network decision functionality, and are positioned to provide faster, more reliable forwarding services.

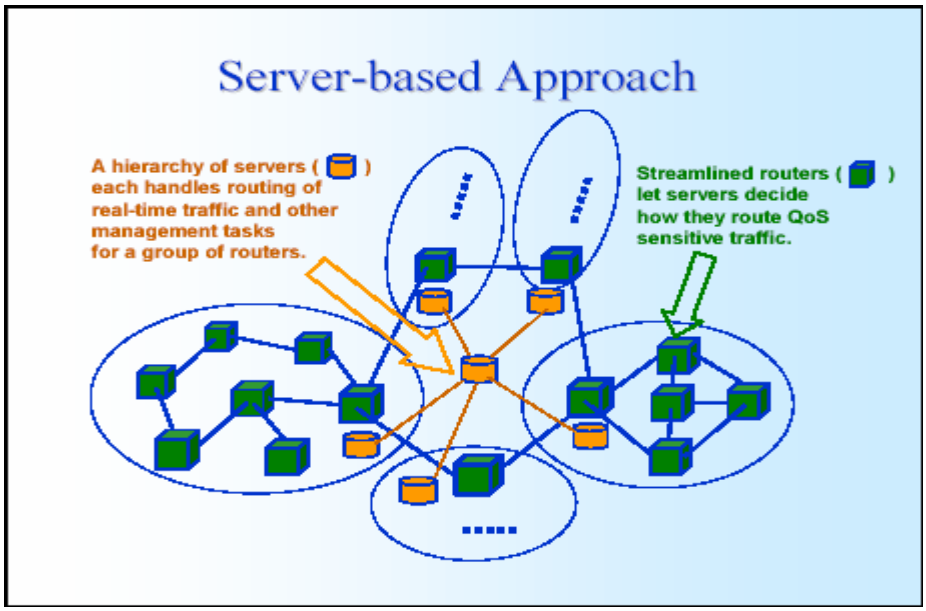

Figure 2. SAAM Server's Responsibility [MAR01]

A SAAM server acts like a helicopter reporting on rush-hour traffic. When a routing request is received, the server (like the helicopter) surveys the state of the

network, determines the best route according to preset criteria, and sends the corresponding routing entries to the routers. Each SAAM server manages an autonomous system, called a SAAM Region. These autonomous systems are also managed by a super SAAM server to control traffic between them (See Figure 3). [MAR01]

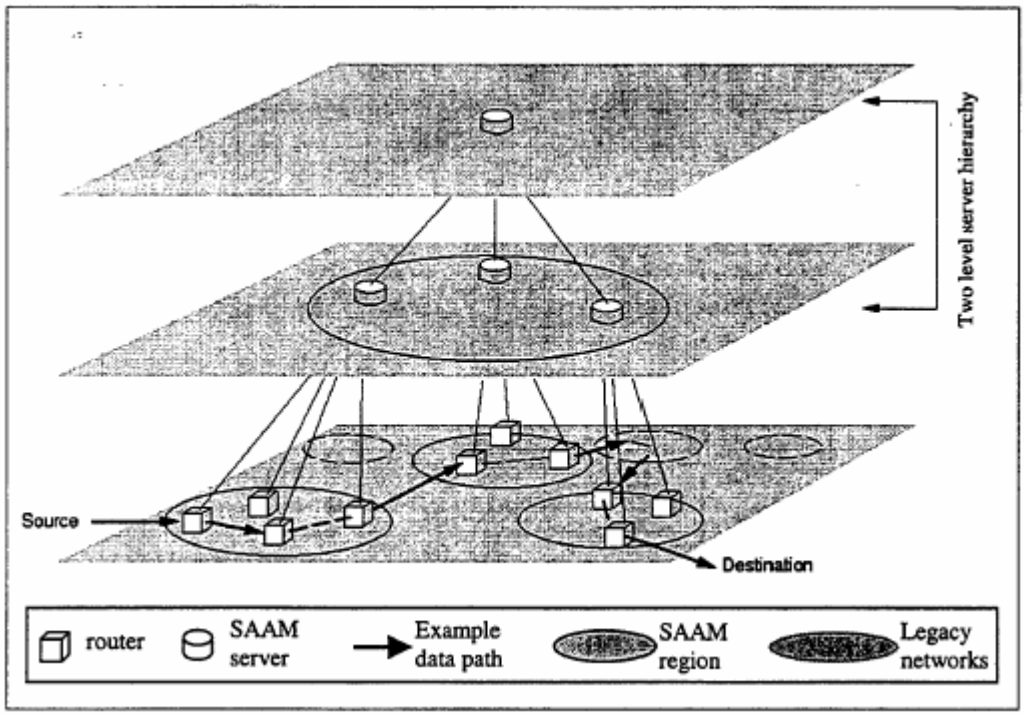

Figure 3. Hierarchical Organization of SAAM servers [KAT00]

The SAAM server needs to collect link state information from all routers in its region. To do so, the server floods a Downward Configuration Message (DCM) periodically to the routers to establish signaling channels for server-router communications. The routers pass up link statement advertisement messages to the server in the process. This flooding is repeated at 200-500 milliseconds intervals so that the SAAM server is able to maintain the near real-time status of the SAAM region (See Figure 4). [MAR01]

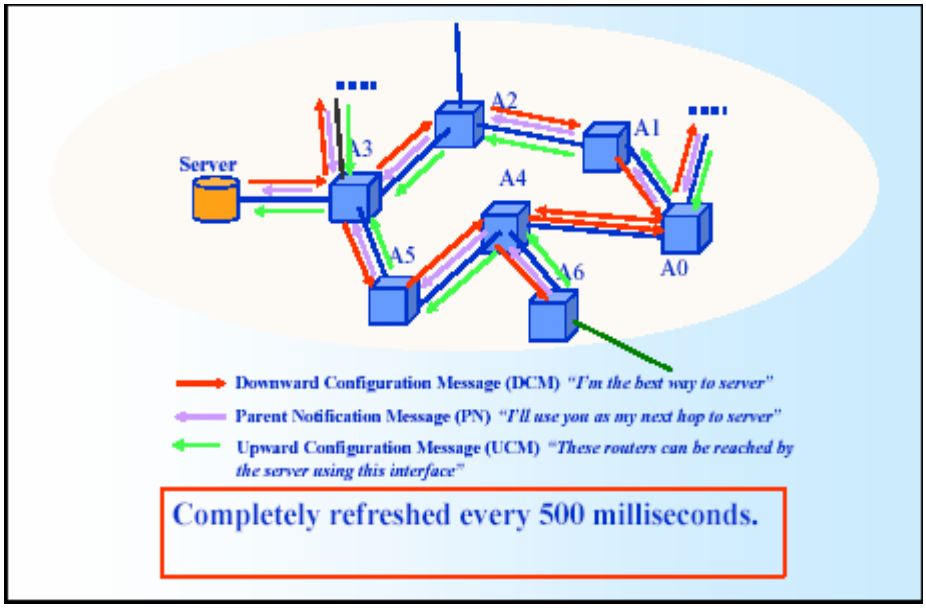

Figure 4. SAAM's Auto Configuration Protocol [MAR01]

2. Brief History of SAAM

Capt Dean Vrable and Capt John Yarger, United States Marine Corps, in their joint thesis, entitled "The SAAM Architecture: Enabling Integrated Services" and published in 1999, laid the foundation of a prototype of the SAAM system. While this foundation was being developed, several other students had already started to study various aspects of the system. Major Brian Tiefert, United States Marine Corps, laid the ground work for SAAM's administration signaling channels and contributed to the development of simulation tools to evaluate the system in his thesis "Modeling Control Channel Dynamics of SAAM using NS Network Simulation". Lieutenant Namik Kaplan, Turkish Navy, completed his thesis, "SAAM Active Server Probing using PLAN and ANEP."

Several major steps were taken by Lieutenant Mustafa Altinkaya, Turkish Navy, first Lieutenant Hasan Akkoc, Turkish Army, first Lieutenant Huseyin Uysal, Turkish Army, first Lieutenant Efraim Kati, Turkish Army, and Mr. Henry Quek, Ministry of Defense of Republic of Singapore, when they turned the SAAM prototype into a more stable system, capable of deploying software agents. The enhancements included an auto-configuration protocol for assembling signaling channels, an efficient mechanism for collecting link state information, and advanced QoS management.

With his thesis entitled “Fault-Tolerant Approach for Deploying Server Agent-based Active network Management (SAAM) in Windows NT Environment to Provide Uninterrupted Services to Routers in case of Server Failure(s),” Lieutenant Kati added the first fault tolerant property to SAAM. He added a backup server to a SAAM region to maintain service in the event of a SAAM server failure.

Captain Luis Velazques, United States Marine Corps, and Peter Szczepankiewicz, United States Navy, developed a scheme for authenticating SAAM control messages with their joint thesis, “Authentication in SAAM routers.” Nevertheless, their work has yet to be integrated into SAAM.

Following this, Lieutenant Colonel Kua Dao-Cheng, Republic of China Army, and Lieutenant Colonel John H. Gibson, US Air Force, worked on maintaining link state information in a more organized data structure with their joint thesis entitled “Design of a Dynamic Management Capability for the Server Agent-based Active network Management (SAAM) System to Support Requests for Guaranteed Quality of Service, Traffic Routing, and Recovery”. The result is an improved Path Information Base (Base PIB).

Lieutenant Mohammad Ababneh, Royal Jordanian Air force, developed a way to vary the SAAM prototype parameters and test topologies and to deploy software agents using the Extensible Markup Language (XML). Lieutenant Fatih Turksoyu, Turkish Navy, added the ability to insert different types of traffic to a SAAM test-bed in his thesis entitled “Realistic Traffic Generation Capability for SAAM Test bed”. Lieutenant Commander Paulo Silva, Portuguese Navy, increased the network utilization with a protocol proposed in his thesis entitled “Advanced Quality of Service Management for next Generation Internet”. This protocol allowed borrowing of resources between integrated and differentiated services. Captain Troy Wright, United States Marine Corps provided a method for efficient re-routing of SAAM packets in the event of a link or router failure. His thesis is entitled “Fault Tolerance in the Server Agent-based Active network Management (SAAM) System.” Lieutenant Cihat Eryigit, Turkish Navy, enhanced the SAAM message channel in his thesis entitled “A Highly Adoptable Generic Event-based Message Channel Design for Loosely Coupling Software Modules”.

Currently, Ltjg. Birol Ayvat, Turkish Navy, Ltjg. Ahmet Guven, Turkish Navy, and Mr. Eng Hong Chua of Singapore, are working on different aspects of SAAM.

B. SURVIVABILITY POSTURE OF SAAM

The SAAM server is such a critical component that the operation of the whole system depends on its survivability. The system can not assure QoS without the proper functioning of the SAAM server. Therefore, the SAAM server needs to be protected to keep the system operative.

Through the evolution of SAAM, many of survivability properties have been integrated into the system to address the need of having a more robust and survivable server. However, the two improvements: the backup server approach in [KAT00] and MAGMA in [MAR01] were the most important ones.

1. Backup Server

The backup server capability was added to the SAAM prototype by first Lieutenant Efraim Kati as noted above. In his thesis, a fault tolerant property for the SAAM server was implemented both locally and remotely. Local fault tolerance for the SAAM server focuses on tolerating component failures of the server, such as processor failure, disk failure, and network interface card failure. For this type of fault tolerance, Lt. Kati investigated some commercial-off-the-shelf (COTS) products and made recommendation.

Unlike local area fault tolerance, the remote area fault tolerance for the SAAM server deals with failures happening outside of the server machine, such as failures due to natural disasters. Lt. Kati proposed using a backup server to tolerate these failures. In this scheme, router nodes are required to send link state updates to the backup server as well as the primary SAAM server so that, the backup server will have the same global view of the network as the primary server. However, the backup server will not respond to router requests until a failure of the primary server is detected.

As a means of detecting the failure of the primary SAAM server, a "heartbeat" protocol is implemented in Lt. Kati's thesis. (See Figure 5)

Lt. Kati's work allows reliable detection of failures of the primary SAAM server. His design, however, assumes that the backup server always remains operative. The backup server may also fail and the failure detection mechanism may not continue to work as expected. More importantly, this fault tolerance scheme cannot effectively tolerate faults caused by an attacker. Since both backup and primary servers are fixed, the attacker will eventually locate these servers, one after the other, and attack them at the same time.

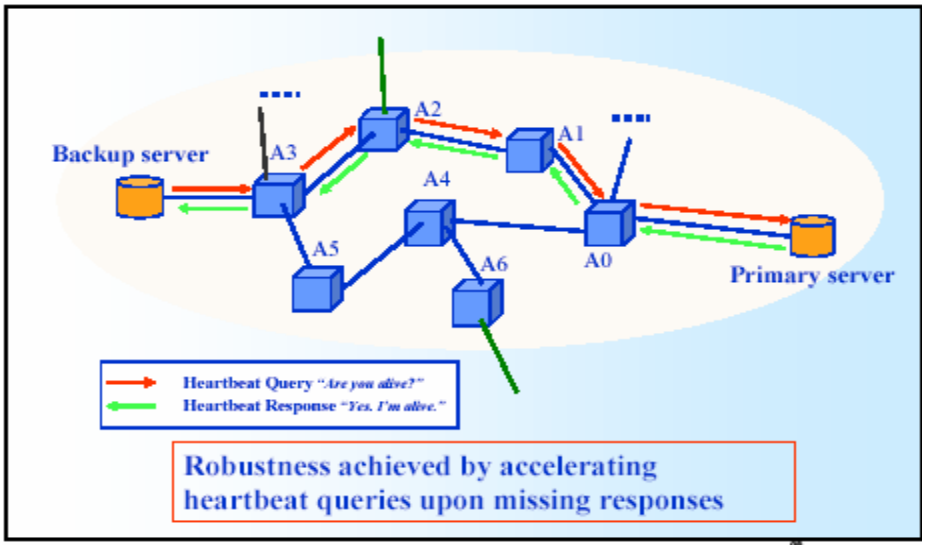

Figure 5. Backup server and heartbeat query [MAR01]

a. Accelerated Heartbeat Protocol

The communication between the primary and the backup server is partitioned into successive time periods. The backup server queries the status of the primary server by sending a heartbeat query message at the end of each period. After that, the backup server starts waiting for a heartbeat response message from the primary server indicating that the primary server is operative. As long as everything goes well and primary server remains operative, the time interval between "heartbeat queries" stays constant (t_{max}). Otherwise the length of the period is determined according to the following rules:

1. The backup server makes the next time-out period to t_{max}, if it receives a heartbeat response message from the primary server within the first half of the current period.
2. If the backup server does not receive a heartbeat response message from the primary server within the first half of the current timeout period, The backup server reduces the next period by half and immediately sends another heartbeat query message
3. If the length of the next time-out period becomes less than t_{min}, a tight upper bound on the network delay between the backup and the primary server, the backup server declares the failure of the primary server [KAT00].

2. The MAGMA Model

The backup server approach certainly improved the SAAM server's fault tolerant posture. However, this idea provided fault tolerance by introducing redundant components. Having the SAAM server static at one location made it more vulnerable to today's sophisticated attacks. Additionally if the SAAM server does not operate properly there will not be any QoS routing decisions provided to routers and the whole system will become useless.

MAGMA added a capability of moving the SAAM server around the SAAM region, with which the survivability of the SAAM server and the whole SAAM architecture improved dramatically. By the time an attacker locates the SAAM server and launches its attack, the server will have already moved to another location, drastically reducing the probability of an attacker's disabling the server. (See Figure 6)

a. The Operation of MAGMA

Margulis Agent-based Mobile Application (MAGMA) uses a programming concept similar to mobile agents. The basic idea behind MAGMA was creating an agent-based SAAM server. An instance of the server agent is installed in the resident memory of each router. The agent remains inactive until it is stimulated by a message from the network administrator or the departing server, in which case the agent begins running a new SAAM server on its host router. The new server starts running from the departing server's most recent state or from a predetermined point if this state information is not available.

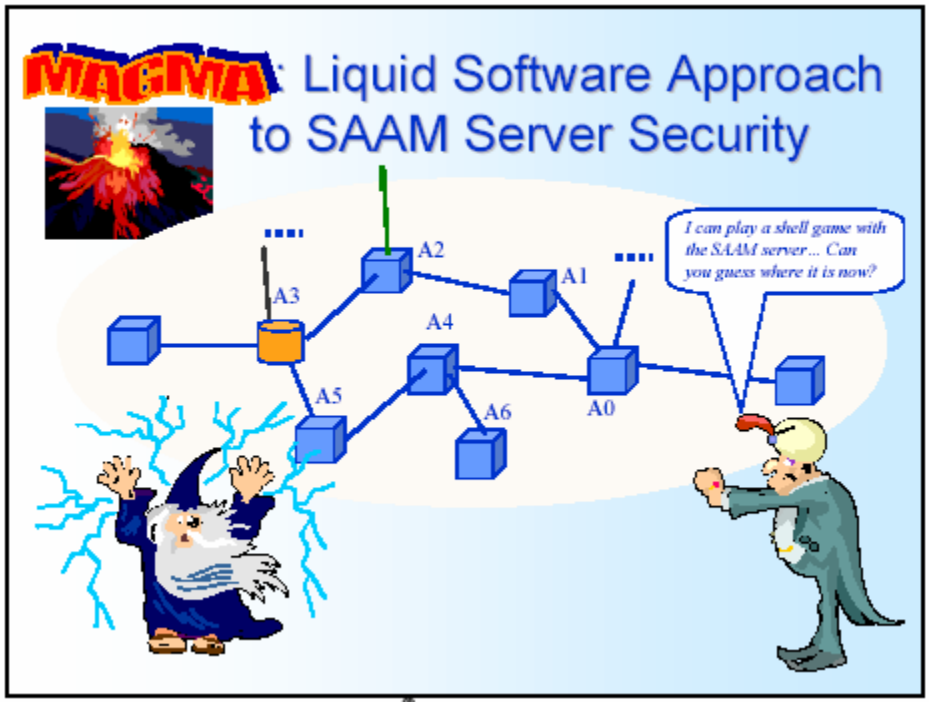

Figure 6. MAGMA' s Security Addition to SAAM [MAR01]

The SAAM server keeps link state information of all its routers in a database called BasePIB. When moving the server from one router to another, it would be desirable to move this database as well. However, sending the entire BasePIB content would tie up too much bandwidth. Therefore, the key state information needed by the next server is taken out of BasePIB and put into a new data structure called DigestPIB. The content of DigestPIB is compressed before transferred to next server host; this reduces transmission time for server state information update, allowing expeditious configuration and setup of the new server. Details about DigestPIB and the compression method can be found in [MAR01].

b. Shortfalls of MAGMA

MAGMA added an essential fault tolerance feature to the SAAM prototype by providing the ability to move the server around the network. Nevertheless, it has two main shortfalls.

First, the movement of the server is done by the system administrator. Second, no metric was defined for selecting the next server candidate. Manual movement

of server is not practical. The movement of the server has to be automated by a protocol and the human factor in the process has to be taken out of the loop.

Second, the questions of "when" and "where" to move the server, were not addressed. A metric for comparing the candidate routers should be developed and a selection protocol has to be built based on this metric. This protocol should make deliberate decisions about when and where to move the server to make the SAAM server most survivable.

To address the second shortfall of MAGMA, this thesis has developed a global connectivity factor to guide server movement and a protocol to automate the selection of the next server host. The factor and the selection protocol will be explained in next two chapters.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. A CONNECTIVITY FACTOR

A. PROBLEM DEFINITION

As explained in Chapter II, solutions proposed by the current studies on survivability lack a widely applicable and quantifiable measure of survivability. These studies usually look at the survivability problem from the software engineering process point of view, which tries to make software agile enough to detect and react to system component failures and software errors. However, these studies seem non-promising regarding practical results in the near future.

Considering the above situation and the problems with the current MAGMA implementation, explained in Chapter III, a strong need for a capability to quantify network survivability has surfaced. This survivability metric should help maximizing the survivability of a network dependent service against persistent network component failures due to attacks and/or natural disasters. The proposed metric should be on a per-client basis. "This level of granularity is necessary because it is often the case that, at a particular moment, a service is more important to some clients than to others" [XIE02].

A simple example can clarify the problem as well as the motivation for this work. Consider deploying the mobile service, explained in Chapter III, over the sample network shown below.

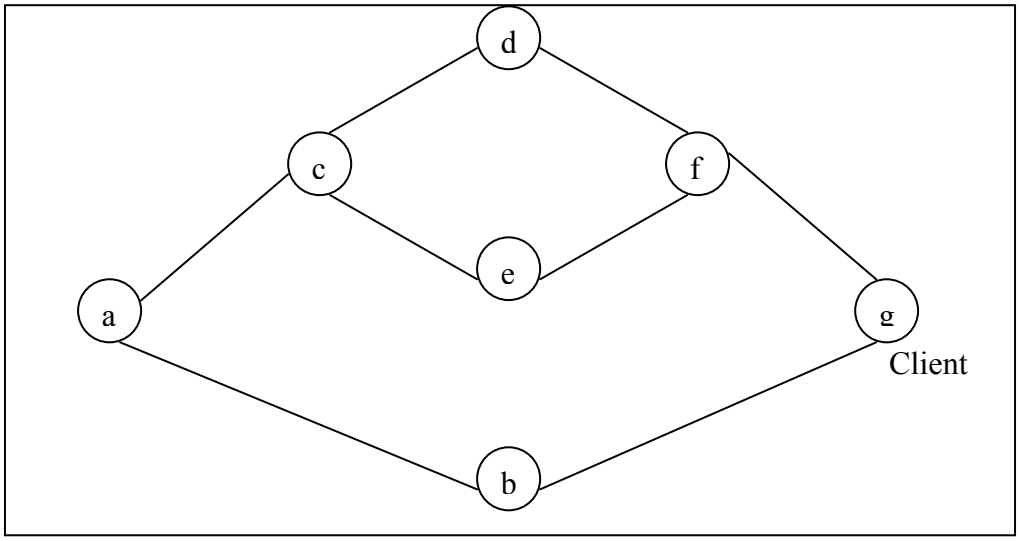

Figure 7. Example Topology [XIE02]

This topology consists of seven nodes, labeled “a” through “g” where “g” is the client node and the other six nodes are the possible hosts for the server. Assuming all links have an independent, uniform probability of failure, the question is whether a difference exists in service survivability when positioning the server at node “a” or “f.” Although the answer is not apparent immediately, it can be obtained as follows. For a single link failure, there is no difference in the client accessing the service since each node has two disjoint paths to the client. However, if two failures occur simultaneously, there is a nonzero probability that the client cannot access the service, in either case. The probabilities are different; because some links are critical in the sense that their failure will cause the number of edge disjoint paths between that node and the client to reduce by one. The client loses connection to the requested service when both links fail.

The set of critical edges on this graph (Figure 7) is {(a-c), (f-g), (a-b), (b-g)}. When “a” is the server location, the probability of losing the service is calculated as (6/7)*(2/6)*2 = 8/42. If the server is located at node “f,” the probability is changed to (3/7)*(1/6)*2 = 6/42. In other words, the service is 25% more survivable if deployed to node “f.” This example shows that, from the point of view of a particular client, certain server locations may provide higher service survivability.

Obviously, the number of edge-disjoint paths is the first criteria for survivability in comparing candidate server nodes. To define the problem more rigorously, let s be the node hosting the service, d be the client, where $s \neq d$, and let Pr {cut(s, d) = 1} be the probability of cut, where cut(s, d) takes the value of 1 if the client can not access the service, and 0 otherwise. If K_e (s, d) is the maximum number of edge-disjoint paths between s and d, the following equation holds:

$$\Pr\{\text{cut}(s, d) = 1\} = \sum_{i=0}^{\infty} \Pr\{\text{cut}(s, d) = 1 \mid \text{i link failures}\} * \Pr\{\text{i link failures}\} \quad (4.1)$$

Since a number of failures that is less than the number of disjoint paths between s and d can not cause a service loss, the above equation can be restated as follows:

$$\Pr\{\text{cut}(s, d) = 1\} = \sum_{i=K_e}^{\infty} \Pr\{\text{cut}(s, d) = 1 \mid \text{i link failures}\} * \Pr\{\text{i link failures}\} \quad (4.2)$$

Typically, the probability of a cut grows rapidly when the number of simultaneous link failures, i, exceeds K_e.

Therefore, $\Pr\{cut(s, d) = 1 \mid K_e \text{ link failures}\} * \Pr\{K_e \text{ link failures}\}$ remains as the critical value for candidate server node ranking comparison purposes. As a result, the left side of equation (4.2), $\Pr\{cut(s,d)=1\}$, can be approximated by $\Pr\{cut(s, d) = 1 \mid K_e \text{ link failures}\} * \Pr\{K_e \text{ link failures}\}$.

Based on the above derivations, the survivability of two server locations s_1 and s_2, with respect to client location d, can be compared in two steps. In the first step, $K_e(s_1, d)$ and $K_e(s_2, d)$ are compared, so the one with a larger number of edge-disjoint paths has the higher survivability.

If the respective K_e values are equal, the comparison has to go to step 2, where the values, $\Pr\{cut(s_1, d) = 1 \mid K_e \text{ link failures}\}$ and $\Pr\{cut(s_2, d) = 1 \mid K_e \text{ link failures}\}$, are computed and compared. The one with the smaller probability has a higher survivability [XIE02].

B. GLOBAL CONNECTIVE FACTOR F_C

The connectivity quality between the server and only one client is considered above. In practice however, the quality of the connections of a service to all its clients needs to be measured. For example, in the SAAM prototype, it is desirable that the new server location has the maximum survivability with respect to all other routers. For this reason, the per-client survivability metric is extended to define a global connectivity factor, F_c, for server location s with respect to a set of clients C, as stated in equation (4.3).

$$F_c(s, C) = \sum_{d \in C} \{R(s,d) * w(d) + e(s,d)\} \tag{4.3}$$

In the definition, R(s,d) represents a ranking of server location s regarding to client d. For example, R(s,d) could just be $K_e(s,d)$ or some other function based on K_e. $e(s,d)$ is used as a tie-breaking mechanism when R(s,d) does not vary much. An example

use of this global connectivity factor will be given in Chapter V where an automated SAAM server relocation process is proposed. In the remainder of this chapter, the focus will be how to compute the important metric K_e.

C. COMPUTING K_e

Finding the number of edge-disjoint paths (K_e) between any pair of nodes on an arbitrary graph takes the first priority in solving the survivability assessment problem. Therefore I delved into the problem of computing the maximum number of disjoint paths and identifying those paths. The problem was not trivial, after doing some research I saw that the problem of finding all possible paths between two nodes on an arbitrary graph is NP complete whereas the maximum number of disjoint paths between two nodes on any graph can be calculated in polynomial time [CAR79]. The argument was promising. Polynomial time complexity was very good since the solution's complexity directly impacts its applicability.

After discovering that the problem of finding the maximum number of disjoints paths between two nodes of any graph was doable in polynomial time, I began searching for ways to solve the problem. Although a solution to the problem, which was developed in the 1970s, was found, the solution did not seem to apply to many practical places. For that reason, finding publications on this problem was difficult. However, the book by Alan Gibbons, "Algorithmic Graph Theory", addressed this problem. Furthermore, in the book, pseudo- algorithms for finding the maximum number of edge or vertex disjoint paths between two nodes of an arbitrary graph were given [GIB85].

1. Algorithm to Find the Edge-Connectivity, Ke (u,v), Between Nodes u and v

a. Some Symbols and Definitions

Cut-set: A cut-set of a connected graph is a set of edges whose removal would disconnect the graph. No proper subset of a cut-set will cause disconnection. In a graph G (V, E) where V is the set of vertices and E is the set of edges, a cut-set can be specified by (P, P') where $\{P\} \subseteq \{V\}$ and P' = V-P

c_e (u, v): The smallest cardinality of those cut-sets that partition the graph such that *u* is in one component and *v* is in the other.

K_e (u, v): Maximum number of edge-disjoint paths between u and v.

G': is a network that is constructed from graph G such that G' contains the same vertex-set as graph G. For each edge, (u, v), of G, G' contains the directed edges (u, v) and (v, u). Each edge of G', e, is assigned a unit capacity, c (e) =1. Thus, for any flow in G', f (e) = 0 or 1.

F (u,v): is the maximum value of a flow from a source, u, to a sink, v, in G'.

Max-flow, Min-cut Theorem: For a given network, N, the maximum possible value of the flow, F, is equal to the minimum capacity of all cuts. Max F (N) = min K (P, P').

If each augmentation only increases the overall flow from u to v by one unit, then the number of augmentations required for maximization would be equal to the min K (P, P').

Theorem [GIB85]: Let G (V, E) be an undirected graph with u, v $\in$ V, then $c_e(u,v) = K_e(u,v)$.

Therefore, when unit weights are used for all edges in G, the following holds: $K_e(u,v) = F(u,v) = c_e(u,v)$.

b. The Algorithm for Finding K_e(u, v)

The following is the pseudo-algorithm for finding K_e

1. Input G (E, V) and construct G'
2. Specify u and v (u, v$\in$V)
3. Find F (u,v) for G'
4. Output $K_e = F$

c. Explanation of the Algorithm

In the first line, the algorithm constructs a diagraph, G', obtained by replacing each edge of G by two anti-parallel edges, each of a unit capacity of 1. The second line is simply identifying the source and the sink nodes. The third line runs the maximum flow algorithm on G'. This line constitutes the major and most time consuming part of the algorithm, so the complexity of the maximum flow algorithm becomes important. Although there are more efficient algorithms, like lift-to-front algorithm with a complexity of $O(V^3)$ this thesis work uses the Ford-Fulkerson's maximum flow algorithm, which has the complexity of $O(VE^2)$. Line four outputs the value returned by the maximum-flow algorithm called on line three.

D. IMPLEMENTATION OF THE ALGORITHM

After discovering how to compute the maximum number of edge disjoint paths K_e, I proceeded to implement the solution. First of all, a program must be implemented to generate graphs so that K_e values can be computed on these graphs. Both graph generation and maximum flow algorithms are very well studied topics. Therefore, a web search was performed to find an existing platform that could satisfy the implementation needs.

Out of many available implementations, an open source Java-based Graph Algorithms Platform (JGAP), publicly available at http://jgap.sourceforge.net11/02 was chosen as the best platform to implement this algorithm.

1. The Java-based Graph Algorithms Platform (JGAP)

The Java-based Graph Algorithms Platform (JGAP) was developed by Ding-Yi Chen, Tyng-Ruey Chuang, and Shi-Chun Tsai of Taiwan. The platform contains a library of common data structures for implementing graph algorithms, allows a user to implement and add new algorithm modules, and includes a performance meter to measure the efficiency of the algorithms implemented. In addition, JGAP allows users to compose computation sequences so that output from an algorithm can be used as input

for another algorithm. It contains built-in implementations of some well-known graph algorithms, such as, Bread-First Search, Depth-First Search, Dijkstra's and Prim's shortest path algorithms, Belman-Ford, and Ford-Fulkerson's maximum flow algorithm.

Two components comprise the JGAP, a graph editing component and a graph algorithm component. The software architecture of JGAP is illustrated in Figure 8.

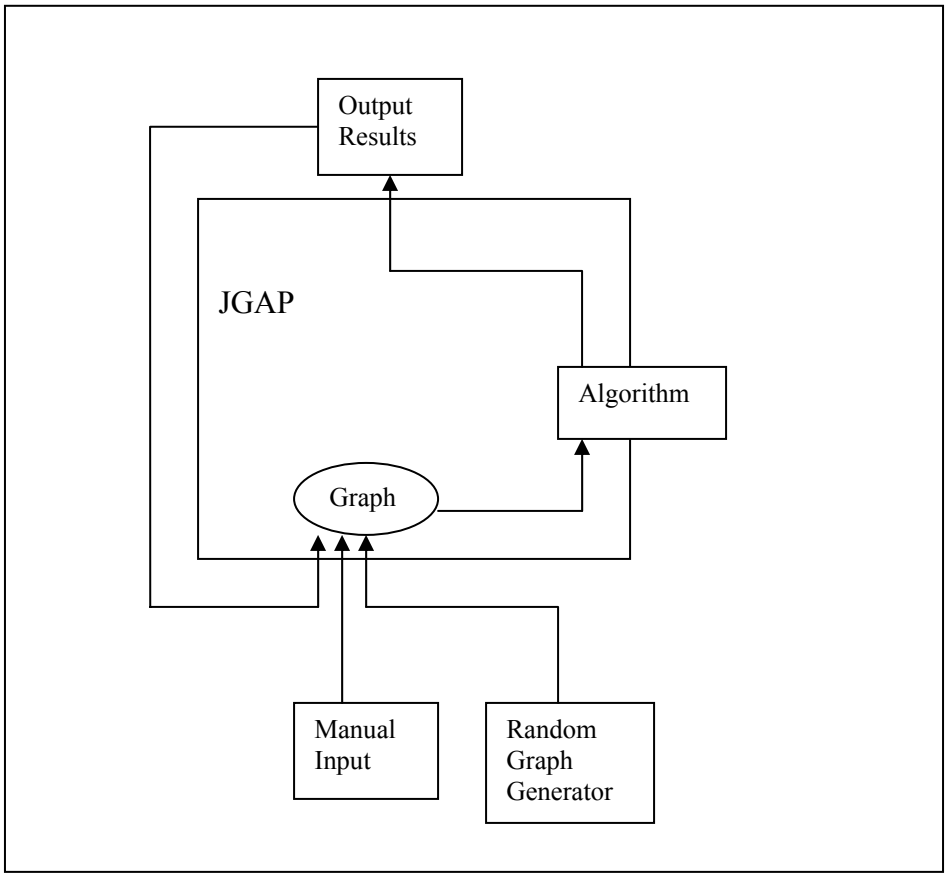

Figure 8. The JGAP Software Architecture [DIN01]

The JGAP graph editing component edits and instantiates graphs from manual input or from the graph generator. Then, graphs are passed to the algorithm component to produce results, which are also graphs and can be fed back to the graph editing component for further processing.

Using the graph editing component, users can add or delete nodes and edges and can view the results. Users can also specify the number of nodes, the edge density, the edge weight distribution, and the type of graph, directed or undirected. Finally, users can decide whether self-loop cycles are allowed while using the random graph generator. The

main window, depicted in Figure 9, pops up when JGAP starts. Its menu bar contains File, Edit, View, Algorithm, History and Help menu items.

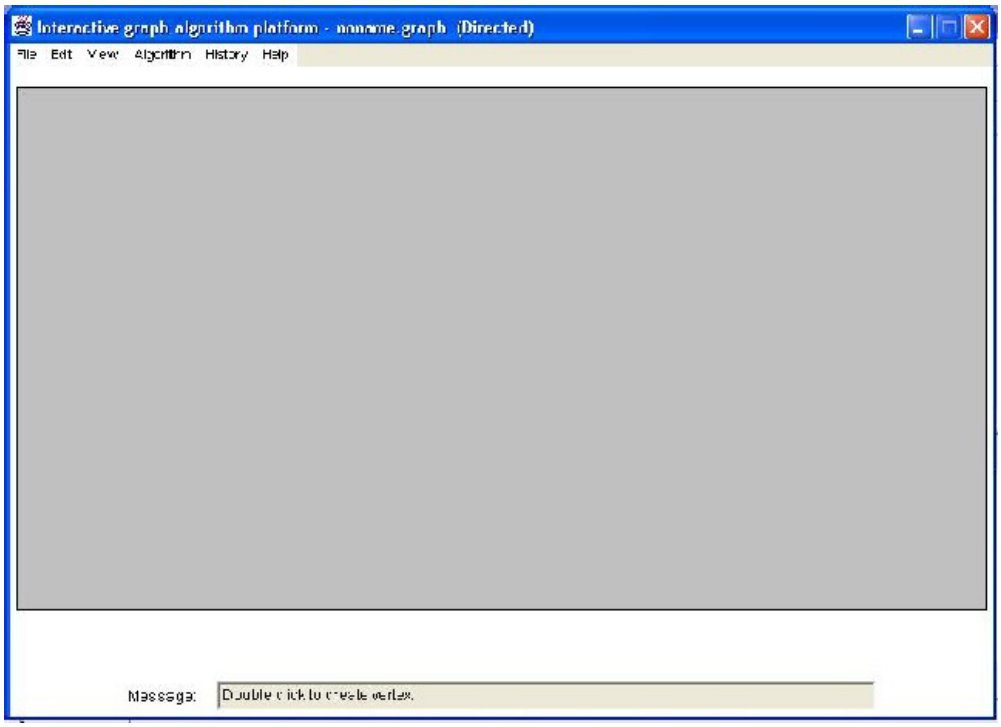

Figure 9. The JGAP main window

a. File Menu

The file menu item allows users to create a graph manually or generate a random graph. The user can also save created graphs and can load these saved graphs in the future.

b. Edit Menu

There are four groups of commands under the edit menu item. The first group allows a user to add or delete edges and change an edge weight. The second group is used to delete or rename the vertices. The third group is used for deleting special edges generated after executing a graph algorithm. The fourth group is used to form the union or difference of two graphs.

c. View Menu

The view menu item allows users to view different representations of the graph, e.g., adjacency matrix, linked-list, or graph.

d. Algorithm Menu

Originally this menu item included many commonly used graph algorithms, like depth-first search, breadth-first search, Prim's algorithm for minimum spanning tree, Dijkstra's algorithm for shortest path, Bellman-Ford's algorithm for shortest path, Floyd Warshall's algorithm for all pair shortest path, and Ford-Fulkerson's maximum flow algorithm. Users can also execute their own algorithm by selecting the custom item, which will load the users' java class library into JGAP.

Since we don't need all of the algorithms available in the original version of JGAP, unnecessary algorithms are taken out and Ford Fulkerson's maximum flow algorithm was modified to address the implementation needs. Thus, only two algorithms are available under this menu item: Find_K_e, and Find_All_Mincuts.

e. History Menu

This menu item allows users to access the graphs that have been generated so far.

f. Help Menu

The help menu item provides basic guidelines on how to use the JGAP platform.

2. JGAP Modified for Finding K_e

Several modifications have been made to JGAP to meet this thesis's implementation requirements. Necessary modifications were made to Ford-Fulkerson algorithm to find the maximum number of edge-disjoint paths, K_e, and the minimum cut-sets (mincuts) between two given nodes. The random graph generator was also modified to generate customized K_e-connected random graphs with uniform edge weight of 1. With this modification, the random graph generator stopped generating very highly connected random graphs, as it previously did, and began to generate K_e-connected

random graphs with the tolerance of +1, meaning randomly generated graphs will always be either K_e or $K_e + 1$ connected.

The new random generator class and new algorithm menu items are explained below. Source code for the added and modified classes is provided in Appendices.

a. Random Graph Generator

The original random generator class in JGAP asked users to determine the edge density, the edge weight lower bound, and the edge weight upper bound. Users also were able to decide on the type of graph to be generated, whether directed or undirected and self-looped or non-self-looped. Since all the edges have to be given a unit edge weight to calculate K_e, the weight lower bound and upper bound fields do not need to be defined by the user. Instead, a non-editable edge weight of 1 is hard-coded. The check boxes for selecting self looped and directed graphs are also removed because self-loops have no effect on maximum flow, and undirected graphs represent the actual network links better. Therefore, a default *false* Boolean value is hard-coded for those selections. Figure 10 illustrates the new random graph generator user interface.

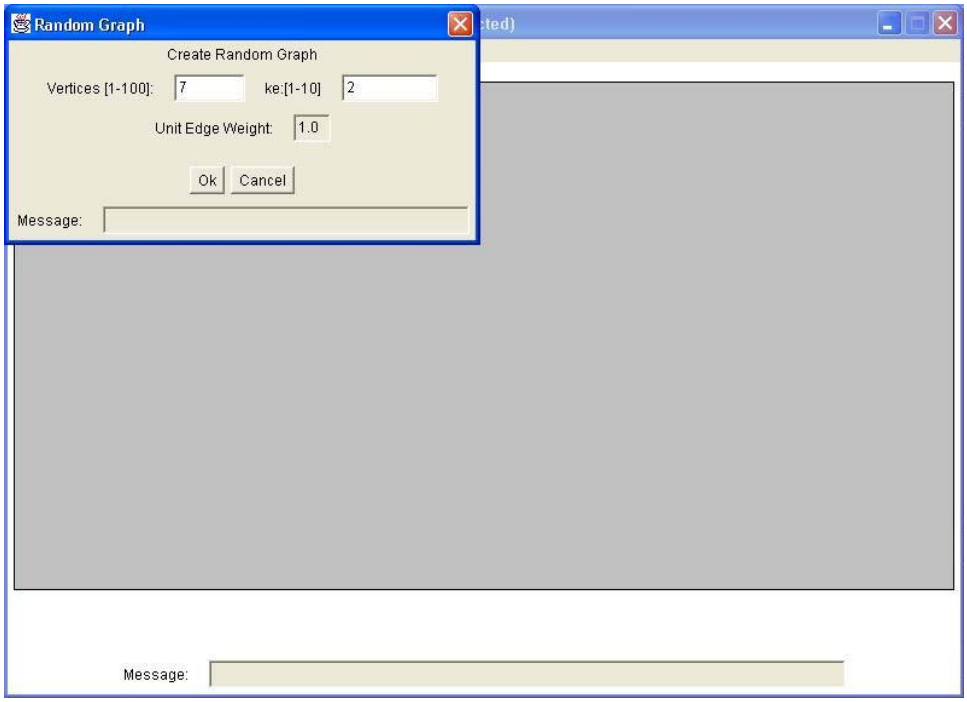

Figure 10. Random Graph Generator

b. Find_Ke Menu

The Find_K_e class is a modified version of the Ford-Fulkerson maximum flow algorithm that is included in the original JGAP platform. This class finds the maximum number of edge-disjoint paths between a user-defined, source and sink node pair on an undirected graph. The algorithm achieves this by assigning a unit flow capacity to every edge of the graph and then running the Ford-Fulkerson maximum flow algorithm on the graph. Since every edge has a unit flow capacity, the returned value by the Ford-Fulkerson Algorithm would be equal to the number of disjoint paths between these nodes. Consider an example graph, generated manually and shown below in Figure 11. The K_e value between vertices 0 and 6 was calculated by the Find_Ke algorithm.

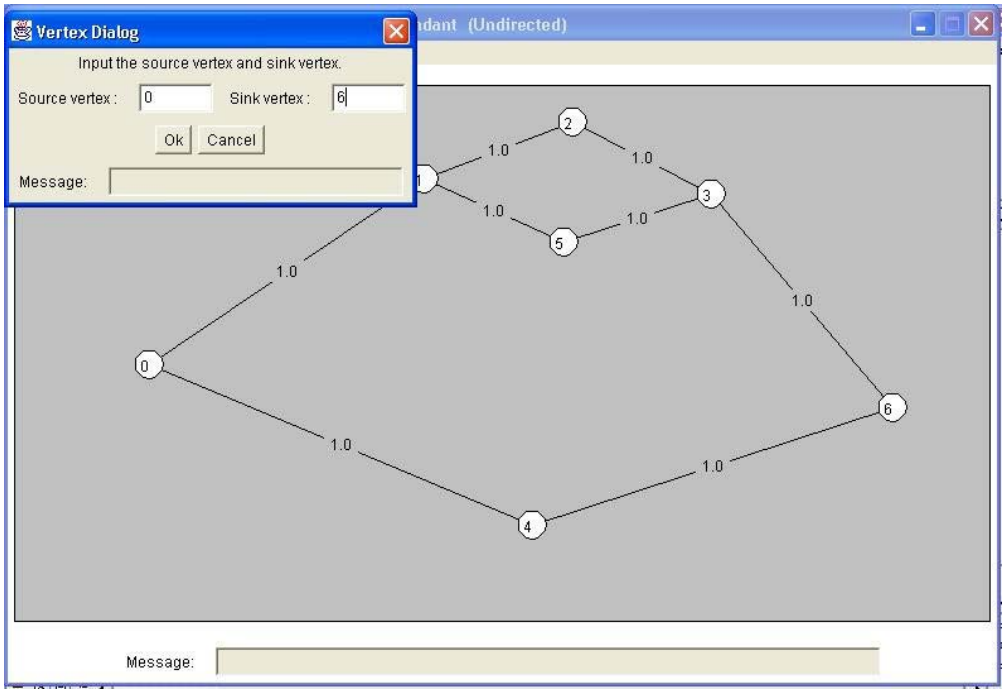

Figure 11. Example Graph to Run Find_K_e

The results are shown in Figure 12. The disjoint paths found are painted with different colors to make them more easily visible. These paths are also identified in a separate window by their vertex sequence from source to destination. It is important to notice that the disjoint paths found by this algorithm are the shortest ones because the

Ford-Fulkerson maximum flow algorithm uses a breadth-first search to find augmenting paths in its execution. There may be alternative disjoint path sets.

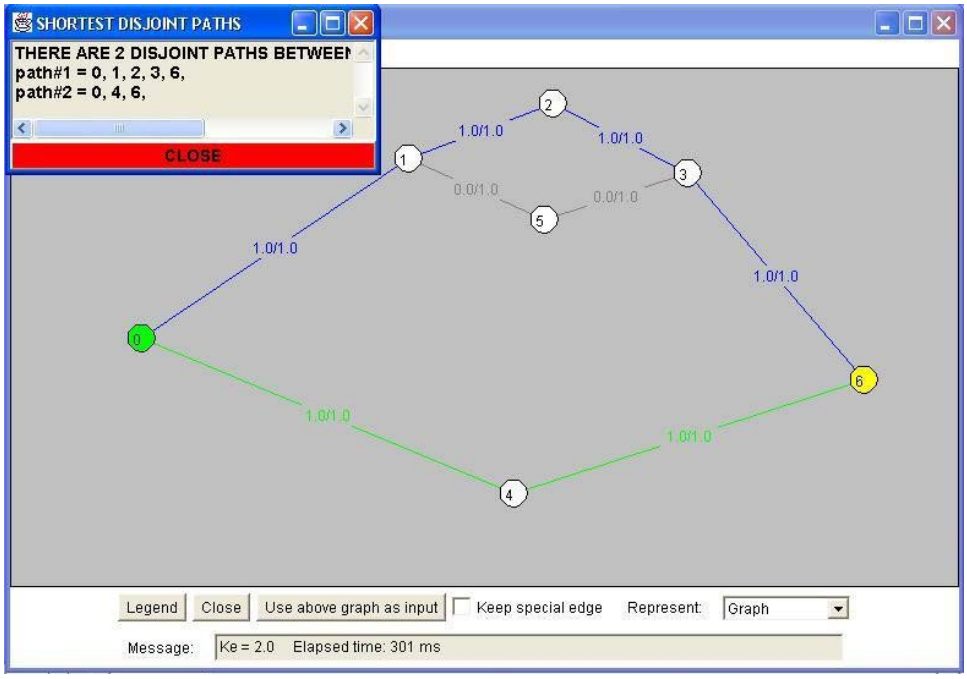

Figure 12. Sample Result of the Find_Ke Algorithm

c. Find All Mincuts Menu

The purpose of finding the maximum number of edge disjoint paths between two nodes of an arbitrary graph was to obtain a metric to compare and rank the server locations with respect to a client. As explained earlier in this chapter, K_e is the first criterion which we propose to use to compare candidate server locations. However, K_e, by itself, typically can not provide the desired level of granularity in comparing and ranking these candidate nodes because the K_e value of more than one node can be the same. In other words a tied condition may occur among some nodes according to their K_e values. In that case, the comparison has to go further. Ideally, the comparison should be based on the number of min-cuts each of the compared node has with respect to the client node because a node with more min-cuts is more likely to be disconnected from the client upon K_e link failures.

Mr. Eng Hong Chua of Singapore, another graduate student at the Naval Post Graduate School, is currently developing a heuristic that will rank the quality of a node's connection to the client without enumerating the min-cuts. To assist testing of his heuristic, this thesis has implemented a brute-force algorithm that can enumerate min-cuts between two nodes with a K_e value up to 4 in a limited size topology. The program is called Find_All_Mincuts and it tests every possible combination of K_e links to determine whether or not they form a min-cut between the node and the client. Specifically, each K_e edge combination is removed and then the Find_K_e algorithm is run on the new graph. If the new K_e value is zero, the edge combination is deemed a min-cut between the node and the client. Unfortunately, this computation is very heavy in complexity. K_e-nested *for-loops* are required to test all K_e edge combinations, which has a complexity of $O(E^{Ke})$. The Find_K_e algorithm has a complexity of $O(VE^2)$, thus making the total complexity of the Find_All_Mincuts program to be $O(VE^{Ke+2})$.

d. Pseudo Code of Find_All_Mincuts Algorithm

```
1.   Input graph G (V, E)
2.   Specify source u and sink v nodes
3.   Ke = Find_Ke(u,v)
4.   Switch ( Ke )
5.    Enumerate all edges of the graph G
6.    Generate Ke nested for-loops such that
7.    For (i=0; i<|E|; i++)
8.        for (j=i+1; j<|E|; j++)
9.        :
10.           for (s=i+Ke; j<|E|; s++)
11.                tempEdge1= E (i)
```

12. ⋮

13. tempEdge K_e = E (s)

14. G_{temp} = G – tempEdges (1… K_e)

15. temp K_e = Find_K_e(u,v)

16. If (temp K_e = 0)

17. tempEdges (1… K_e) are critical

18. Else tempEdges (1… K_e) are non-critical

19. G_{temp} = G_{temp} + tempEdges (1...K_e)

20. Output K_e tuples of min-cuts

This algorithm works in a brute force fashion. The Algorithm takes the graph G (V, E), and calculates K_e for specified source and sink nodes, *u* and *v*. Then the algorithm enters a switch statement based on this K_e value. K_e nested *for-loops* are created to test all possible K_e edge combinations. The following process is repeated for each K_e edge combination: Edges of the combination are stored in a temporary location, and then removed from the graph G. After that, K_e is recalculated on the new graph. If this K_e value is equal to 0, then those edges constitute a min-cut. All min-cut edges are stored in a data structure for future access. Temporary edges are added back to the graph. The algorithm outputs all min-cuts in a separate window at the end of its execution. Figure 13 and 14 are the screen shots from running the Find_All_Mincuts program. Notice that the numbers located on the edges are different in Figure 13 and Figure 14. The ones in Figure 13 stands for the unit weight whereas the ones in Figure 14 show the edge identities. Thus, the user can identify the min-cut edges on the graph more easily.

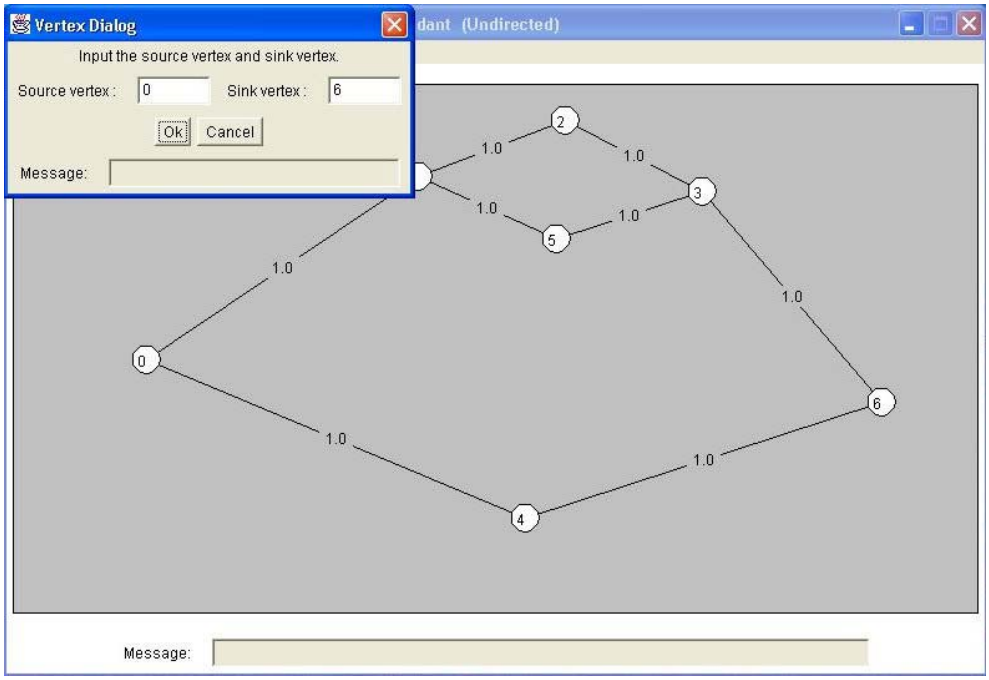

Figure 13. Example Graph to Find All Mincuts

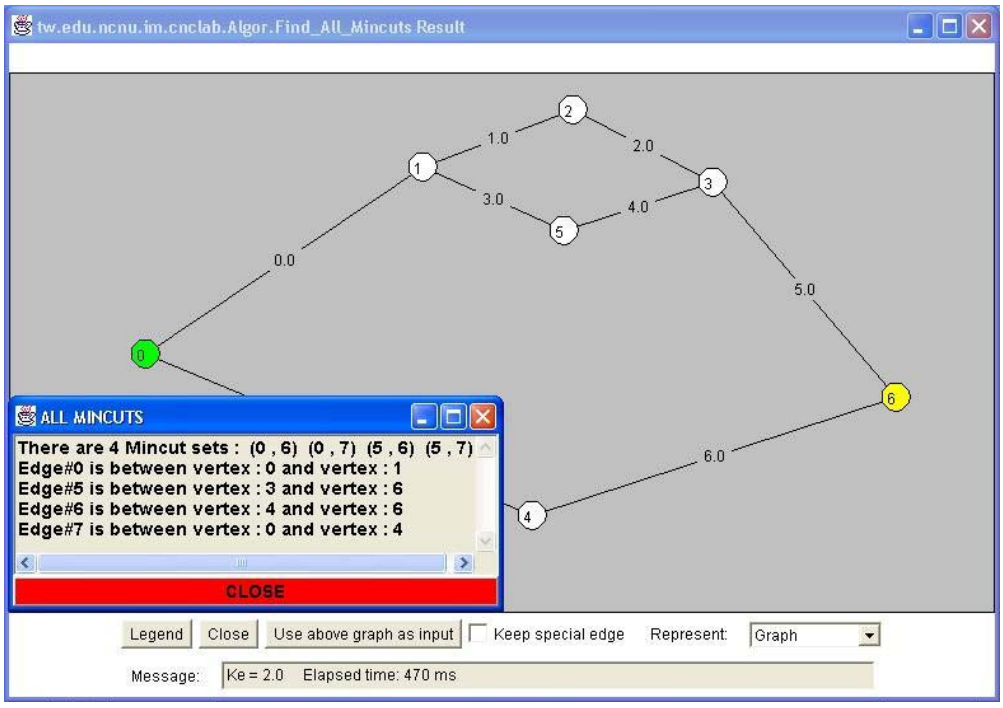

Figure 14. Result of Find All Mincuts Algorithm

THIS PAGE INTENTIONALLY LEFT BLANK

V. REVISED FSI MODEL

As stated in Chapter I, an ad hoc FSI model was developed in a class project [AKT02] to address the remaining survivability problems of the SAAM prototype after MAGMA integration. This model was developed in a very limited amount of time and it was not based on strong theoretical foundations. This chapter first describes this old model and its shortfalls. It then describes enhancements made to the FSI model by this thesis.

A. OLD FSI MODEL

The FSI model was developed to capture many factors in moving the SAAM server. This model proposes to relocate the SAAM server in a random fashion to minimize detection and the probability of attacks against it. The model also recommends equalizing the sizes of all SAAM control messages and encrypting them. The recommendation is mainly intended to address packet-sniffing types of attacks.

The idea is: if an attacker probes the network and sniffs the links, he may capture and examine packets in transit. If all control messages are sent in clear, then the attacker can easily collect information from these messages and locate the server node. Therefore, the model proposes to encrypt control messages to hide the location of the SAAM server. Even though the contents of the messages are encrypted, there is still a chance for the attacker to distinguish the types of messages according to their sizes. Therefore, all control messages should be of the same size after encryption.

1. The FSI Metric and Its Calculation Policy

The main concern of the model was moving the SAAM server around the network in a random and efficient manner. But a metric has to be defined to help determine “when” and “where” to move the server. The model proposes to use a dynamic metric called Fit-to-Serve-Index (FSI) to rate the suitability of a router for hosting the SAAM

server at a certain time. The FSI metric was mainly derived from two factors: reliability and probability of attack.

$$FSI = (Reliability) / (Probability\ of\ Attack) \quad (5.1)$$

In other words, at a given time a candidate router with a higher reliability and a lower probability of attack would have a higher FSI value resulting in a higher probability of hosting the SAAM server.

The reliability factor was derived based on several properties of hardware and software components. Furthermore, these properties were given weight factors (wf_x) to reflect their importance.

$$Reliability = \{(wf_1 * Hardware\ Scale) + (wf_2 * Software\ Scale)\} / (wf_1 + wf_2) \quad (5.2)$$

The probability of attack value was derived similarly. However, the deciding factors were physical location, the number of people that have direct access and the number of interfaces the router has.

$$Probability\ of\ attack = \{(wf_3 * Physical\ Location) + (wf_4 * Number\ of\ People) + (wf_5 * Number\ of\ Interfaces)\} / (wf_3 + wf_4 + wf_5) \quad (5.3)$$

The hardware and software scales were further decomposed into finer levels of granularity. The hardware scale was derived from three factors: firewall, hardware random number generator (R.N.G.), and system quality.

$$Hardware\ Scale = \{(wf_6 * Firewall) + (wf_7 * Hardware\ R.N.G.) + (wf_8 * System\ quality)\} / (wf_6 + wf_7 + wf_8) \quad (5.4)$$

The software scale was determined based on the strengths of firewall, software random number generator, O/S quality, and encryption algorithms.

$$Software\ Scale = \{(wf_9 * Firewall) + (wf_{10} * Software\ R.N.G.) + (wf_{11} * O/S\ quality) + (wf_{12} * Encryption\ Algorithm)\} / (wf_9 + wf_{10} + wf_{11} + wf_{12}) \quad (5.5)$$

Each of the lowest level factors was given a rating by the network administrator based on a fixed scale. For example, the firewall factor can have a scale as shown in

Table 2 below. The scales for other factors will not be explained here since they are all ad hoc. However, they can be found in [AKT02].

Average	0.4
Good	0.7
Excellent	1.0

Table 2. Firewall Scale

The model proposes to decrease the FSI value of the router currently hosting the server functionality by a fixed hourly factor until the FSI value drops below a preset threshold, in which case the server functionality must be relocated. This provides a mechanism for the model to move the server under normal working conditions, i.e., when there is no intrusion. The model also proposes to increase the FSI value of the routers that has just relinquished the server responsibility by the same hourly factor until the FSI value reaches a preset maximum. The hourly factor is defined as the ratio between a predetermined $FSI_{Threshold}$ and the maximum server hosting time for a router which is denoted by T_{max}

$$\text{Increase Factor} = \text{Decrease Factor} = FSI_{Threshold} / T_{max} \quad (5.6)$$

Once the FSI values for all the routers are computed the system works according to the execution flow defined below.

a. Model Execution Flow

1. When one of the routers takes over SAAM server functionality, it broadcasts an FSI-Request packet in order to obtain the FSI values of all other routers in the same SAAM region.

2. Upon receiving an FSI-Request packet, all other routers reply with their FSI parameters. The SAAM Server maintains a FSI table and broadcasts the content

of this table after a random period between 20 - 60 minutes to make it more unpredictable from the perspective of traffic analysis.

3. This action is repeated randomly until the SAAM Server's FSI value goes below $FSI_{Threshold}$

4. When the SAAM Server determines that its host's FSI value is less than $FSI_{Threshold}$, it takes the following actions:

a) If the number of routers in the region is less than 5, selects the next SAAM Server host randomly from all of the routers

b) If the number of routers in the region is between 6 and 10, selects the next SAAM Server host randomly from the $\lceil n/2 \rceil$ routers in the region that have higher FSI values.

c) If the number of routers in the region is between 11 and 20, selects the next SAAM Server host randomly from the best $\lceil n/3 \rceil$ routers based on their FSI values.

d) If the number of routers in the region is over 20, then selects one randomly from the best $\lceil n/4 \rceil$ routers based on their FSI values.

Since all routers in the SAAM region keep a FSI Table, in case of a successful denial of service (DoS) attack on the SAAM Server or an unanticipated hardware failure, the server agent on one of the functioning routers, one with the highest FSI value, will activate a new SAAM Server. This action is invoked after the routers miss DCM messages from the server for 3 seconds. Recall that the server floods a DCM message to all routers at time intervals of several hundred milliseconds, as described in the beginning of Chapter III. If no DCM message is generated for another second, the server agent on the router with the second best FSI will try to activate itself as the SAAM Server. This action will be repeated until a new SAAM Server becomes operational and sends out DCM messages.

The model execution flow for both server node and the ordinary router node are explained below in Figure 15 and Figure 16 with finite state machine diagrams.

b. Server Node Finite State Machine Diagram

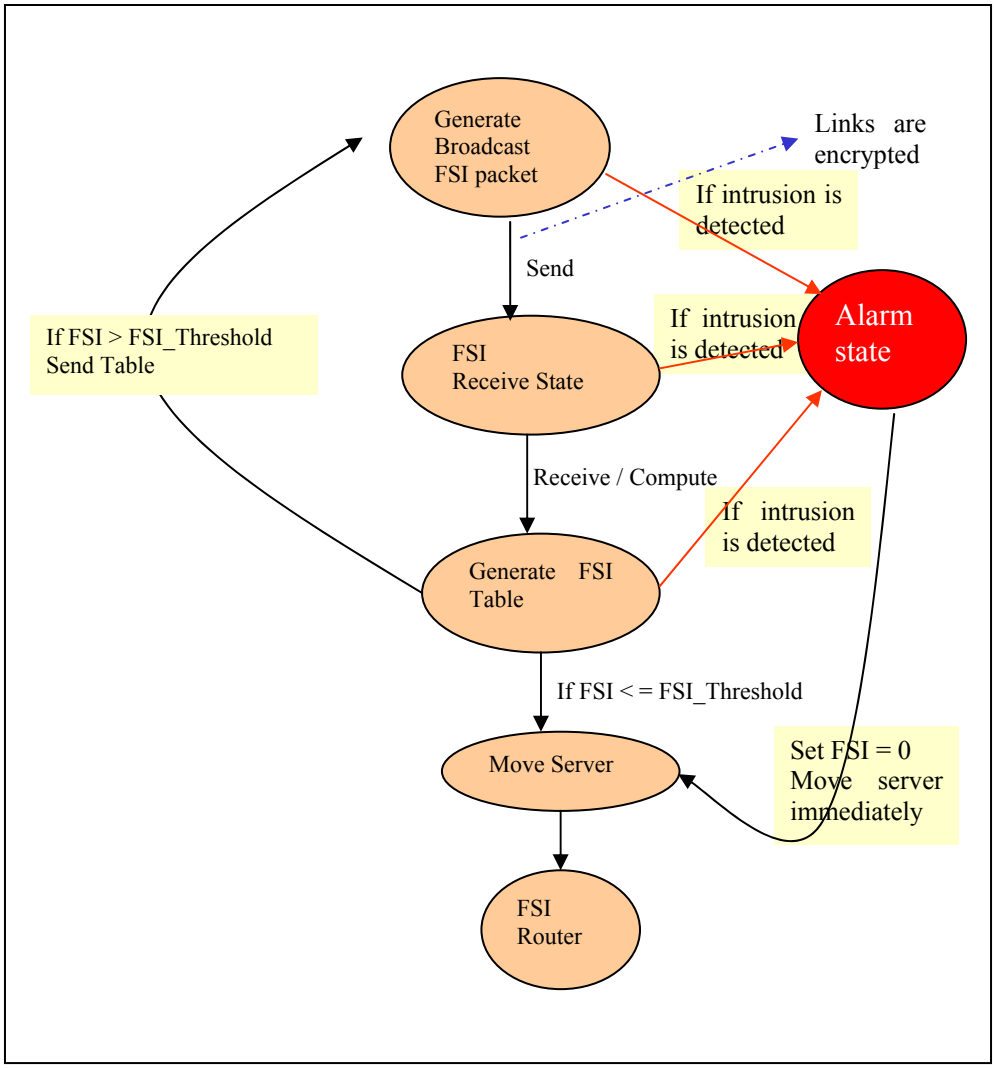

Figure 15. Server Node Finite State Machine Diagram [AKT02]

c. Router Node Finite State Machine Diagram

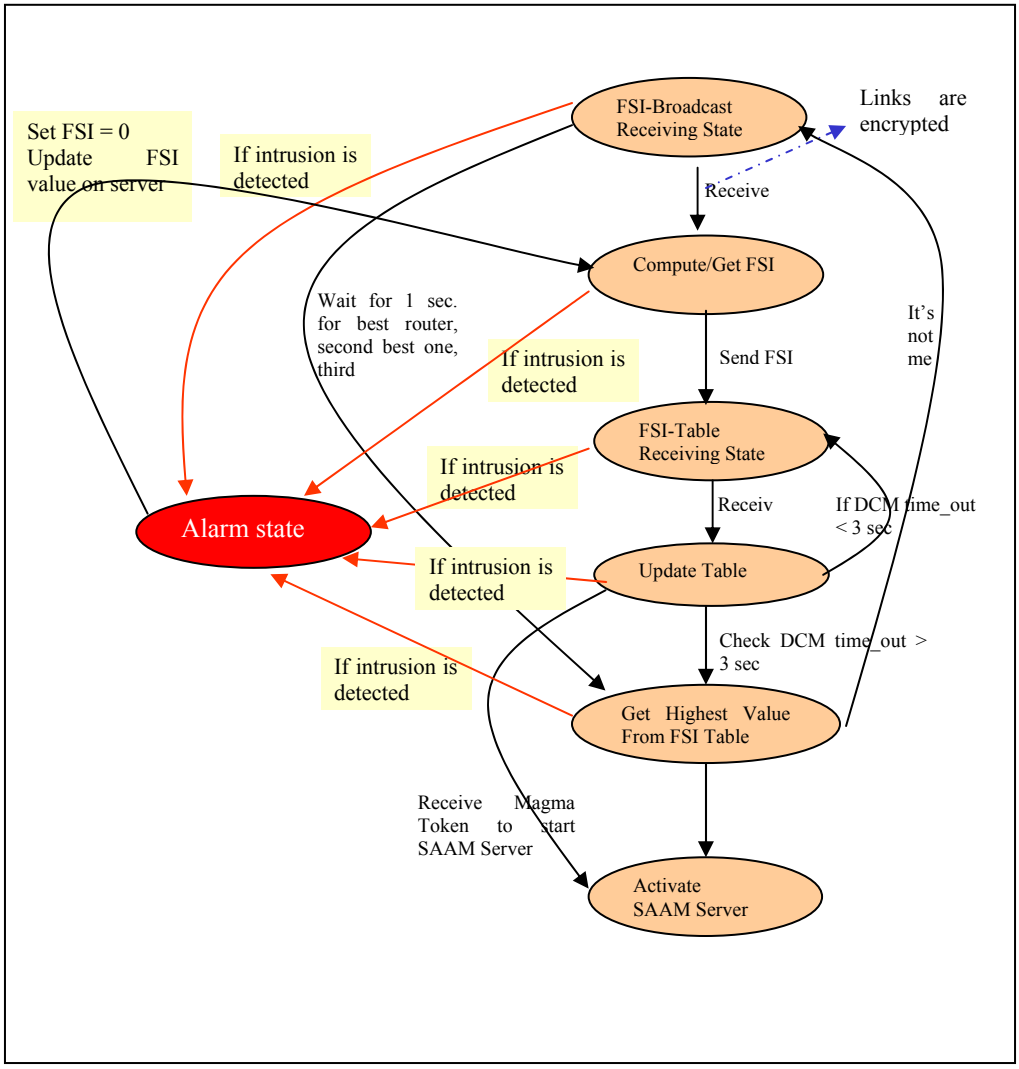

Figure 16. Router Node Finite State Machine Diagram [AKT02]

2. Shortfalls of the Old FSI Model

The major shortfall of the old FSI model is that it is ad hoc. None of the factors or scales used in the model was based on strong theoretical foundations. The model was missing concrete metrics like the global connectivity factor, F_c, defined in Chapter IV. Use of encryption in control messages to hide the server node location from an attacker was another aspect of the system to be reconsidered. The processing complexity caused by the encryption may cause the system to overload the router nodes, which totally contradicts to the main goal of the SAAM architecture.

The model was not designed to handle all possible failure combinations. For example, if the server node fails to distribute the FSI table to the other router nodes at the very beginning of the system execution, then there will not be any FSI table available for the routers to check their FSI values against and the system will not initialize properly.

Additionally, in this model if an intrusion is detected on any of the routers, the FSI value of that router is immediately dropped to zero. This mechanism may cause the model to fail under concerted denial of service types of attacks. With most of the routers under attack and their FSI values dropped to zero, there would be very few or no candidate routers to move the server, in which case the model execution will be meaningless.

Another problem with this model is that the sum of the FSI values of the candidate routers that are in the selection pool tends to decrease with time. And this decrease becomes sharp when there are intrusions on these routers. As a result of the intrusions the FSI values of these routers will be dropped to zero and new routers with lower FSI values will have to be put in the selection pool. Eventually, all available routers in the selection pool will have FSI values a little above $FSI_{Threshold}$. In that case, the server would begin to jump from one location to another at small time intervals, which can also be considered as an unstable condition.

In the following section, a revised FSI model addressing these shortfalls will be explained.

B. REVISED FSI MODEL

1. Rules of the Model and Assumptions

a. Each router maintains a FSI table. A router sends update of its FSI to the server node, which then redistributes the update to all other routers.

b. Server node determines its successor router node as soon as it starts the server. The routers are notified of the selection by a special FSI update message.

c. Server node will distribute its FSI value as part of the DCM message for control purposes, such as resolving the conflict caused by multiple concurrent servers.

d. A predetermined $FSI_{Threshold}$ value is known by all routers. The server determines the number of routers that will be in the candidate routers pool after it receives the first FSI update messages from the routers.

e. If more than one router has the same FSI value and there is not enough space in the candidate routers pool for all of them, the one that is not in the FSI increasing state will be the first choice. This information will be sent to the server as a binary value (0= not increasing, 1= increasing) as part of the FSI update message and will be stored in the FSI table. If there is still a tied condition, the one that hosted the server earlier than the others will be selected. To achieve this, there will be a time value associated with each router on the FSI table showing the last time that the server is hosted by that router. This time value will be zero initially and it will only be modified by the router as soon as it starts running the SAAM server.

f. It is assumed that there exists a robust and effective intrusion detection mechanism on each router.

g. When a router detects an intrusion, it will not drop its FSI value to zero immediately; instead, the FSI value will be decreased linearly as long as the intrusion is on. The FSI value will be increased multiplicatively when the intrusion is over according to the intrusion detection mechanism. Here the term multiplicative increase means that, the FSI value will be doubled every hour.

h. Server will be moved when one of these two conditions is true:

I. The server is hosted for a predetermined maximum service time, t_{max}, or

II. The FSI value of the server host goes below $FSI_{Threshold}$

i. If an intrusion is detected on a router, this router notifies the server with a special message and begins decreasing its FSI value linearly. As soon as the intrusion is over it starts increasing its FSI value multiplicatively. The router updates the server node about these new FSI values by sending periodical special FSI update messages.

j. If the server node receives a special message from its successor requesting not to host the server, it removes this router from the candidate routers pool, selects a new successor router, updates the FSI table and distributes the updates to all other routers.

k. If the server node receives a special message from an ordinary router, it removes this router from the candidate routers pool if it is in the pool, updates the FSI table and distributes it to all other routers.

l. If the router crashes before it can send the special message or if the special message is lost in transmission, the server determines the failure of the router after several DCM cycles. In that case, the server sets that router's FSI value to zero, and removes it from the candidate routers pool.

m. If the server crashes at the moment it receives the special message but before it can make the necessary updates

I. If the successor node is still up and running, it will take over the server responsibility after missing DCMs for a predetermined amount of time.

II. If the successor node also crashes before it can take over the server responsibility, the reactive election protocol determines the next server.

n. If the server crashes down at the very beginning of the process before it distributes the FSI table to other nodes. The other routers will discover this situation from missing DCM messages and each router will generate a random back off time to declare itself as the server node. A router will declare itself as the server at the end of this random period of time unless it receives a DCM message by then. But there is a non zero probability that more than one router will declare itself as the server. To resolve this conflict:

I. If there is only one server running, the normal procedure explained above will be used

II. If there are more than one server running at the same time, the FSI value that is sent with the DCM messages will be used to resolve this conflict. If a server node receives a DCM message with a higher FSI value from another server node, it will

quit from being server node. Similarly, a router only responds to the DCMs with maximum FSI value.

o. The routers for the candidate routers pool are selected from the ones that are not under intrusion. That's why there may be a situation that the current number of routers that are in the pool is less than the number that is determined by the pooling policy. If there is no router left that is free of intrusion, then the ones with higher FSI values are selected for the candidate routers pool.

The execution flow of the revised FSI model is depicted with finite state machine diagrams in Figure 17, Figure 18, and Figure 19. The Boolean variable "old" used in Figure 18 is initially set to *false*. And it determines the flow of execution for the router node when there is an FSI table available.

2. Server Node Finite State Machine Diagram

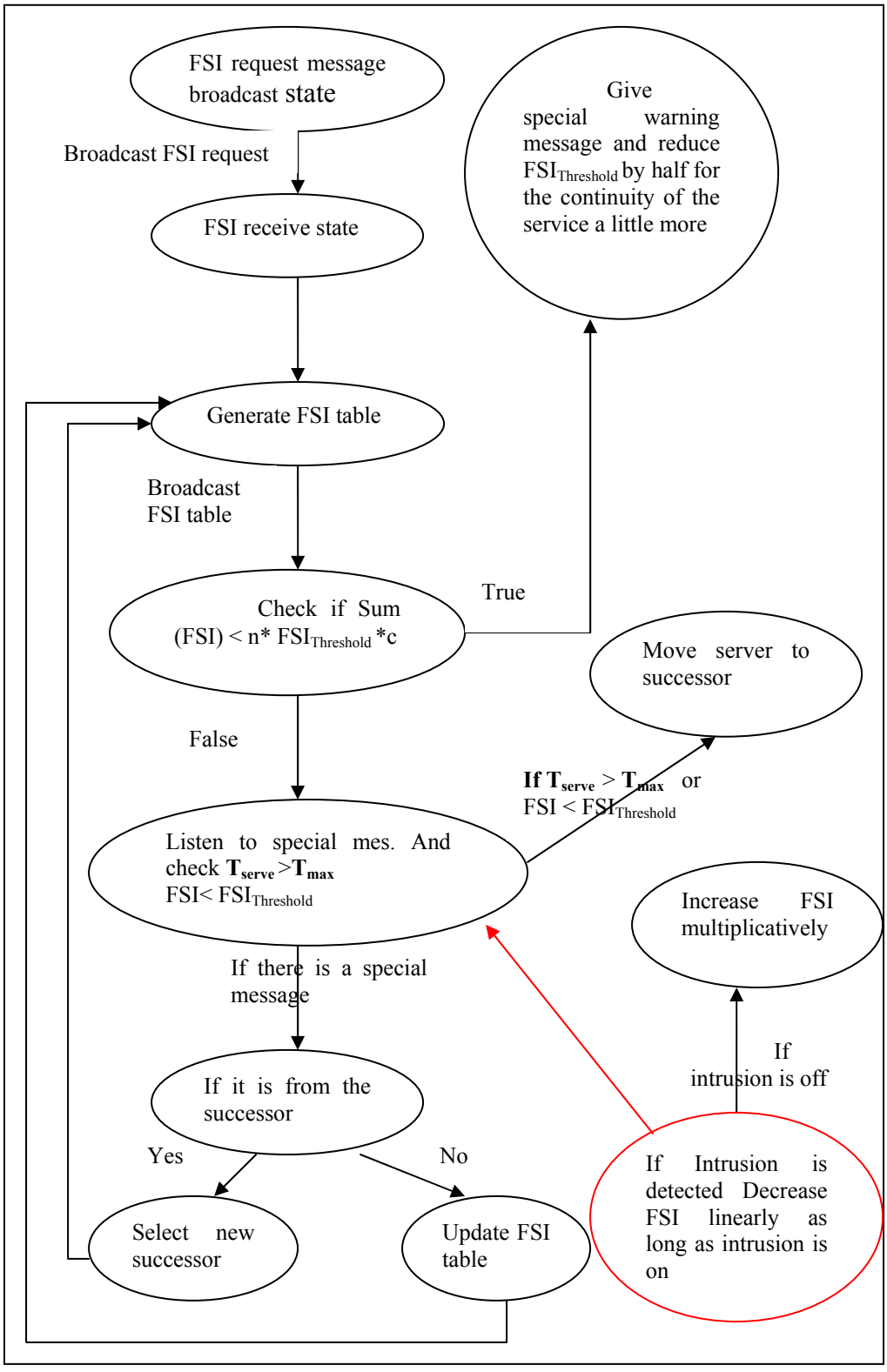

Figure 17. Revised FSI Model Server Node Finite State Machine Diagram

3. Router Node Finite State Machine Diagram

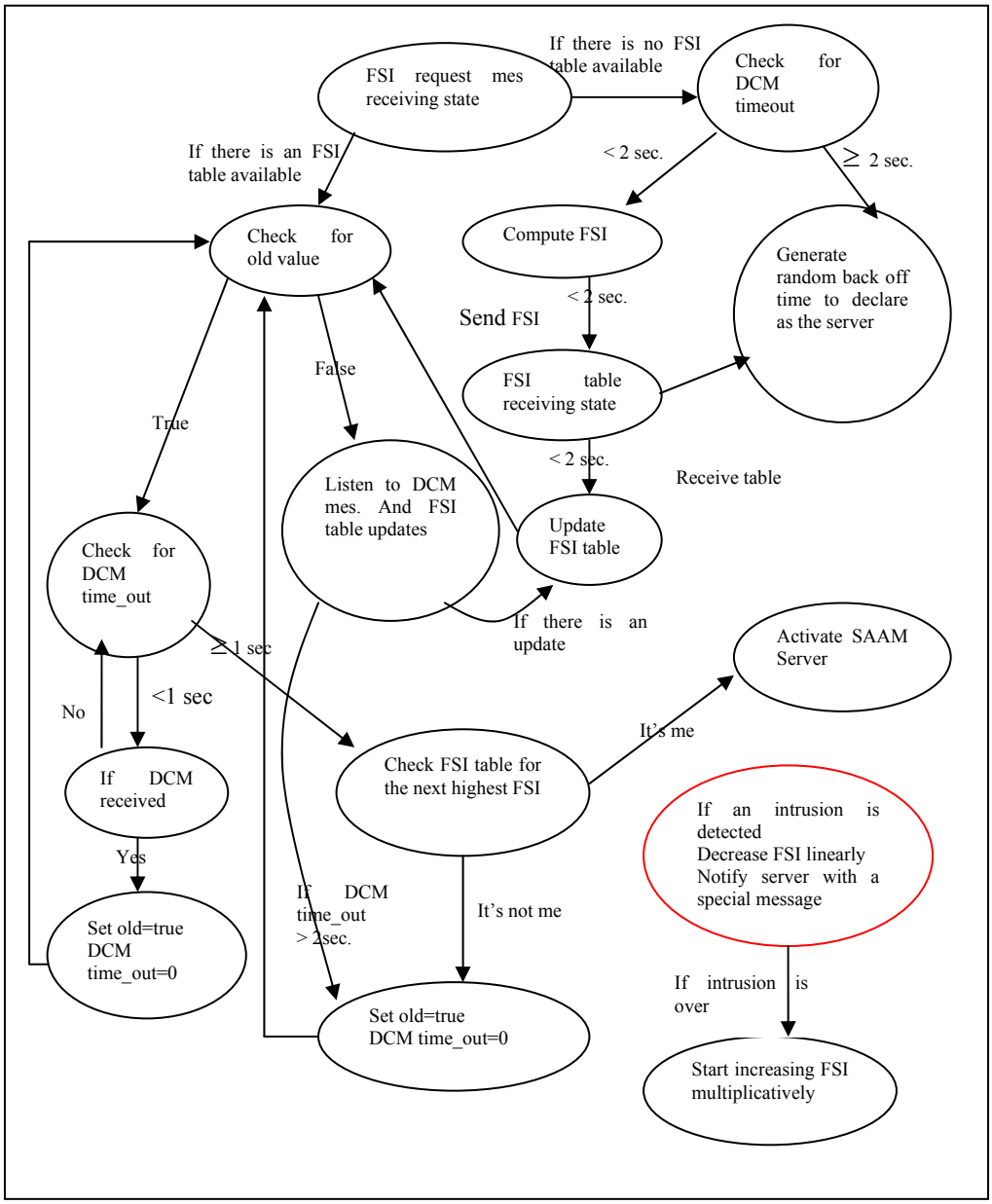

Figure 18. Revised FSI Model Router Node Finite State Machine Diagram

4. Successor Node Finite State Machine Diagram

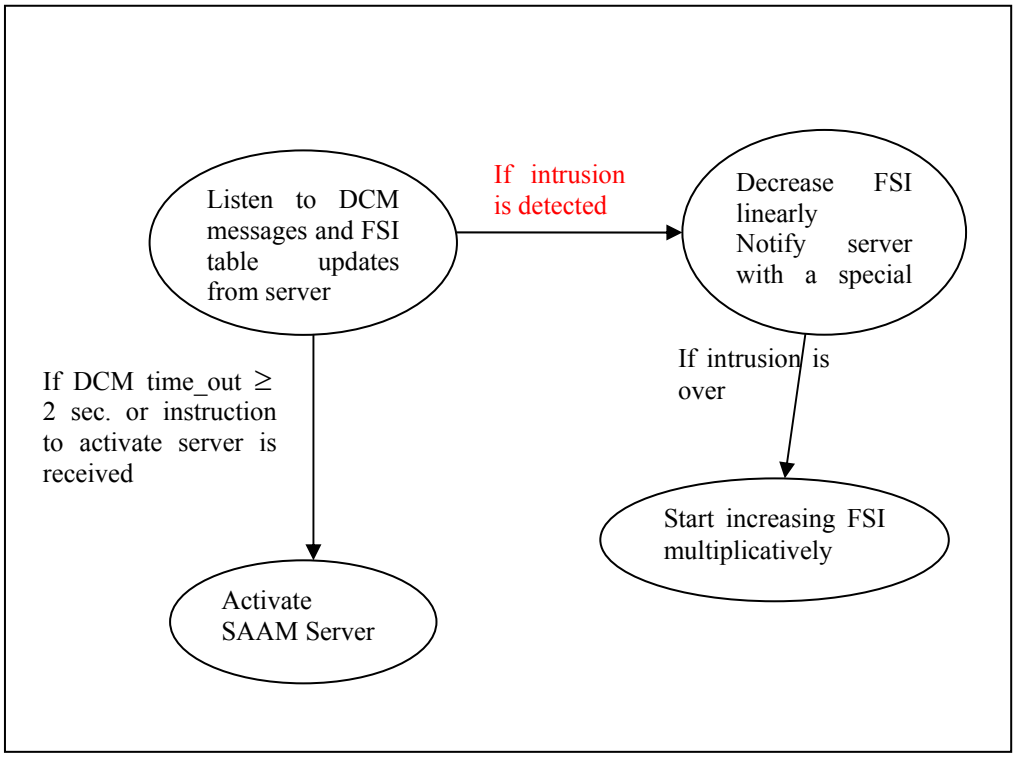

Figure 19. Revised FSI Model Successor Node Finite State Machine Diagram

5. The Improvements By the Revised FSI Model

As already stated, the major problem of the old FSI model was its lacking of reliable metrics. Although, the old model certainly has several good metrics e.g., software scale, hardware scale, those are not included in the revised model because they are ad hoc. Instead, the global connectivity metric F_c, as defined in Chapter IV, is used to determine the FSI values of the routers. First, the node, for which we are calculating the FSI value, will be considered as the server node while all other nodes will comprise the set of client nodes. Second, a ranking factor $R(s, C)$ is defined for the server node based on the connectivity metric K_e. Recall R(s,C) is an important component of F_c.

Although this ranking method results in tied FSI values for the nodes that have the same K_e and weight factors; it still provides a good rough level ranking for many situations. However, the comparison must go further when there are tied ranking conditions. The Find_All_Mincuts algorithm could help to break these tied conditions by considering the exact number of minimum-cut-sets between the server and the client

node. The FSI equation could be modified as follows if the complexity of Find All Mincuts Algorithm were not exponential.

$$FSI(s) = F_c(s,C) \tag{5.7}$$

$$R(s,d) = c \cdot K_e(s,d) \tag{5.8}$$

$$e(s,d) = \frac{1}{n_c(s,d)} \tag{5.9}$$

Where $n_c(s,d)$ represents the number of minimum cut sets between server node s and client node d; $w(d)$ is the weight factor of the client d indicating its importance, and c is a constant greater than 1, which is used to intensify the effect of $K_e(s,d)$.

Unfortunately, the complexity of Find_All_Mincuts algorithm was too heavy to be practical. For this reason, Mr. Eng Hong Chua is working on the development of a heuristic that will substitute the role of the Find_All_Mincuts algorithm. His heuristic seems promising in giving correct ranking decisions in polynomial time. But, his heuristic is not considered in this thesis due to time constraints.

Nevertheless, the revised FSI model will determine FSI values of each node with reliable and concrete metrics after integration of the heuristic explained above. Additionally, unlike the old FSI model, the revised FSI model considers every possible state that the system might enter. The model can handle various types of server failures, concurrent servers, and a failure of the successor node.

Another problem with the old FSI model was that the system was not designed to work under heavy denial of service types of attacks. And the reason for this weakness was that the old FSI model immediately drops the FSI value of the intruded router to zero. The new model does not drop the FSI value immediately to zero but decreases it linearly as long as the intrusion is on. Thus, making the model more resistant to denial of service attacks.

The revised FSI model also resolves the problem of decreased sum of FSIs of the routers that are currently in the candidate routers pool. The revised model achieves this by decreasing FSI linearly when the intrusion is on and increasing FSI multiplicatively

when the intrusion is off. The TCP congestion protocol is taken as the model when integrating this mechanism into the revised FSI model. This mechanism not only prevents to the total FSI from continuously decreasing but also tolerates false alarms of the intrusion detection mechanism.

In summary, the revised FSI model addresses the problems of the old FSI model and provides a more rigorous and complete way of moving the SAAM server around the network. An example with some possible scenarios will be given below to demonstrate the workflow of the system.

6. Example Execution of New FSI Model

The topology shown in Figure 20 will be used in this example.

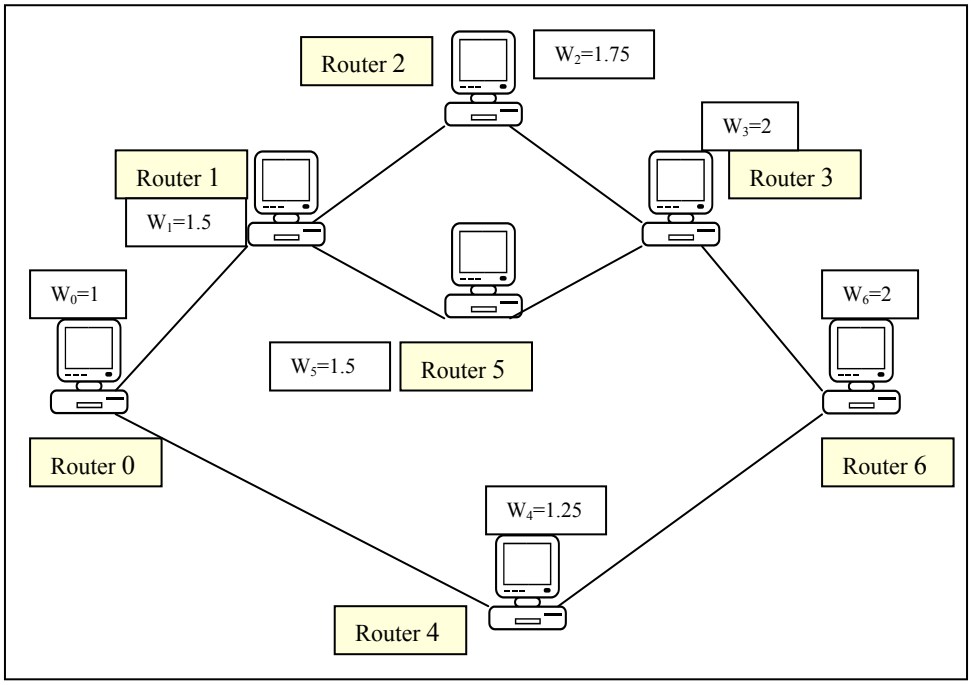

Figure 20. Example Topology

The following are major parameters used in the example:

The weight factor of each node as listed in Table 3

K_e value for each node pair as listed in Table 4

n_c value for each node pair as listed in Table 5

Initial FSI values computed using equation (5.7), as listed in Table 6.

Node Id	0	1	2	3	4	5	6
Weight Factor (*w*)	1	1.5	1.75	2	1.25	1.5	2

Table 3. Weight Factors of the Nodes in the Example Topology

Node Id	0	1	2	3	4	5	6
0	X	2	2	2	2	2	2
1	2	X	2	3	2	2	2
2	2	2	X	2	2	2	2
3	2	3	2	X	2	2	2
4	2	2	2	2	X	2	2
5	2	2	2	2	2	X	2
6	2	2	2	2	2	2	X

Table 4. K_e Values for the Example Topology

Node Id	0	1	2	3	4	5	6
0	X	3	4	3	3	4	4
1	3	X	1	16	4	1	3
2	4	1	X	1	5	2	4
3	3	16	1	X	4	1	3
4	3	4	5	4	X	5	3
5	4	1	2	1	5	X	4
6	4	3	4	3	3	4	X

Table 5. Mincut Values for the Example Topology

Node Id	0	1	2	3	4	5	6
Initial FSI	101.75	117.95	95.7	100.45	99.05	98.2	91.75

Table 6. Initial FSI Values for Each Node In the Example Topology

The constant factor c used in equation (5.8) is set to 5 when computing the initial FSI values. The other required parameters for the model to execute are listed below.

$FSI_{Threshold} = 30$

$T_{min} = 20$ hours

$T_{max} = 30$ hours

Increase factor = Decrease factor = 5 (per hour)

The size of candidate routers pool = $\lceil 7/2 \rceil = 4$

Router Id	FSI Value							
	Initial	**2.5 sec.**	**3 sec.**	**5 sec.**	**30 min.**	**22 hr.**	**32 hr.**	**33 hr.**
0	101.75	101.75	101.75	101.75	101.75	101.75	101.75	101.75
1	117.95	117.95	117.95	117.95	117.95	117.95	117.95	117.95
2	95.7	95.7	95.7	95.7	95.7	95.7	95.7	95.7
3	100.45	100.45	100.45	100.45	100.45	100.45	100.45	100.45
4	99.05	99.05	0	0	99.05	99.05	99.05	99.05
5	98.2	98.2	98.2	98.2	98.2	98.2	98.2	93.2
6	91.75	91.75	91.75	91.75	91.75	91.75	91.75	91.75
Sum (FSI)	-	-	-	514.05	517.4	517.4	514.9	514.9

Table 7. Revised FSI Model Example Execution Flow Table

	Current server node
	Successor router node
	Routers that are currently in the candidate routers pool
	Router node currently under intrusion
	Server node currently under intrusion

Table 8. Color Codes used by Table 7 and Table 9

The execution sequence is depicted in Tables 7 and 9. Different color codes are used to represent the different states of a node. The color codes are explained in Table 8. The specific events and actions taken by the new FSI model are as follows. The system is assumed to start execution at time 0.

At time 0: The model starts with router 4 as the server node. However, router 4 crashes as soon as it starts, even before sending an FSI table to the other nodes.

At 2 seconds: After not receiving a DCM within 2 seconds, all other routers do a random back off to determine when to declare themselves as the server unless they receive a DCM by the end of the back-off period.

At 2.5 seconds: Router 1 and router 6 declare themselves as the server simultaneously because they have the same back-off period.

At 3 seconds: Router 6 receives a DCM messages from router 1 with a higher FSI value. Router 6 returns to a normal router mode. The FSI value of router 4 is set to zero since no control message is received from that node for several DCM cycles.

At 5 seconds: The FSI table is generated and distributed to other nodes. Router 1 is the server host; router 3 is the successor; routers 0, router 2, 3, and 5 are in the candidate routers pool.

At 30 minutes: Router 4 has recovered and updated its FSI value to the server. As a result the candidate routers pool is also updated. Now router 4 is added to the pool and router 2 is removed from the pool.

At 22 hours: Router 3 becomes the server and selects router 5 as its successor. Router 0, router 1, and router 4 are in the candidate routers pool.

At 32 hours: An intrusion is detected on router 5. Router 5 notifies the server about this intrusion with a special message and starts decreasing its FSI value linearly as long as the intrusion is on. Upon receiving this special message, the server removes router 5 from candidate router pool, selects a new successor router, updates the FSI table and distributes the updates to other nodes. Router 5 continues to update the server about its FSI value with periodical special messages.

At 33 hours: An intrusion is detected on router 3. This server node starts decreasing its FSI value linearly. The intrusion on router 5 is still on.

At 35 hours: The server node has crashed. After missing DCMs for 2 seconds the successor, router 0, declares itself as the server and selects router 6 as its successor. The intrusion at router 5 is off at 35h 02 sec. Router 5 starts increasing its FSI value multiplicatively. The FSI value of router 3 is set to zero.

At 36 hours: Both server and the successor nodes fail simultaneously. After missing DCMs for 3 seconds, router 1 declares itself as the new server since it has the highest FSI value in the FSI table. At time 36 h 05 sec, router 1 is the server, and router 4 is the successor. Router 3 then recovers and updates its FSI value to the server.

Router Id	Initial FSI	35 hours	35 h 02 sec	36 hours	36 h 03 sec	36 h 5 sec	38 hours	40 hours	61 hours
0	101.75	101.75	101.75	~~101.75~~	101.75	0	101.75	101.75	101.75
1	117.95	117.95	117.95	117.95	117.95	117.95	117.95	117.95	117.95
2	95.7	95.7	95.7	95.7	95.7	95.7	95.7	95.7	95.7
3	100.45	~~90.45~~	0	0	0	100.45	100.45	100.45	100.45
4	99.05	99.05	99.05	99.05	99.05	99.05	99.05	99.05	99.05
5	98.2	83.2	83.2	88.2	88.2	88.2	98.2	98.2	98.2
6	91.75	91.75	91.75	~~91.75~~	91.75	0	0	91.75	91.75
Sum (FSI)	-	504.9	506.2	506.2	-	501.35	517.4	517.4	517.4

Table 9. Revised FSI Model Example Execution Flow Table Continued

At 38 hours: Router 0 recovers and updates its FSI value to the server. Router 5 achieves the maximum FSI value as a result of multiplicative increase.

At 40 hours: Router 6 recovers and updates its FSI value to the server node.

At 61 hours: Router 4 becomes the server and selects router 3 as its successor. Routers 0, 1, and 5 are in the candidate routers pool.

The last row of both Table 7 and Table 9 shows the total FSI value of the candidate routers pool. It is included to verify the model's stability. The results indicate that the new model is stable.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS

A. SYNOPSIS AND CONCLUSIONS

The goal of this thesis was to develop a model for maximizing survivability of network services, specifically of the SAAM prototype.

The SAAM prototype was made more survivable with the integration of the mechanism called MAGMA designed by Scott Margulis [MAR01]. MAGMA's capability of moving the SAAM server around the SAAM region, dramatically improved the survivability of the SAAM server, and the whole SAAM architecture. However, this mechanism was unpractical since it was not automated. The movement of the server should be automated and moreover this movement decision must be intelligent.

To address those problems stated above, appropriate metrics must be developed to quantify network survivability and an automated election protocol must be designed for the movement of the SAAM server. Therefore, a global connectivity metric, K_e, which represents the maximum number of edge-disjoint paths between two nodes of an arbitrary undirected graph, was developed in this thesis.This metric provided a means to quantify and compare network survivability on a per-client basis. Thus, K_e allowed us to compare the survivability of candidate server locations of the SAAM server.

Additionally, an election protocol for the automated and yet intelligent movement of the SAAM server was designed in this thesis work. The developed model used the metric, K_e as the dominant factor in comparing the survivability of the candidate server locations. And it comprised of three different types of components: the SAAM server, the successor router, and the ordinary router. All failure combinations of the components are aimed to be handled by the model. Finally, the execution flow of the model is explained by an example with possible attack scenarios.

To sum up, I accomplished the basic intended goals for this thesis work. However, there are many areas of future work remained open for new researchers due to the potential wide range of network survivability and its immaturity. Some of these potential future work areas are listed in the next section.

B. FUTURE WORK

An effort to maximize network survivability was made by quantifying survivability in this thesis work. But, this was only considered to be a beginning for the process of maximizing network survivability. Undoubtly, neither the creative minds of future researchers nor the list of future works for this thesis can be limited. Nevertheless, some potential future work areas are listed below.

1. Develop Survivability Metrics Other Than K_e

A global connectivity metric K_e was developed to quantify network survivability in this thesis. But, this was only one metric out of many potential network survivability metrics. Survivability builds on related fields of study, which include security, fault tolerance, safety, reliability, and performance. Therefore, new metrics related with these fileds can be developed and integrated into FSI metric.

2. Metrics in the Old FSI Model Can Be Studied

The factors, defined in the old FSI model such as hardware scale, software scale, probability of attack were ad hoc and they were not included in this thesis work. Nevertheless, these factors can always be good metrics for network survivability. Studying these factors and integrating them into the FSI model can be another future work.

3. Implementation and Integration of the Model

The developed model for the autometed and intelligent mevement of the SAAM server was neither implemented nor integrated to the current SAAM architecture. Implementation and integration of the model can be another future work.

APPENDICES

The whole source code of JGAP platform will not be given. Instead, the classes that are added and modified will be listed in Appendices. Complete source code can be obtained by contacting xie@nps.navy.mil.

A. CLASSES ADDED TO JGAP

1. Find_All_Mincuts.java

```
package tw.edu.ncnu.im.cnclab.Algor;

import tw.edu.ncnu.im.cnclab.UI.*;

import tw.edu.ncnu.im.cnclab.DataStru.*;

import tw.edu.ncnu.im.cnclab.Tools.*;

import tw.edu.ncnu.im.cnclab.JGAP.*;

import tw.edu.ncnu.im.cnclab.JGAP.UI.*;

import java.util.*;

import java.awt.*;

/**

*This class finds critical edges or edge tuples on the criteria of computed Ke of given 2 nodes of a *given graph, which represents a network topology. The procedure for finding all mincuts: First of *all Ke is computed between given two nodes, and then every Ke combination of all the edges of *the graph is deleted and Ke is recomputed on the new graph. If new Ke is equal to zero, then *these deleted edges are identified as a minimum cut set. After this deleted edges are added back *to get the original graph. These steps are repeated for every Ke combination and the results are *displayed in a seperate window.

*@Author Baris AKTOP

*Modified for finding all Mincuts betwwen two nodes of an arbitrary undirected graph

*March 2003

*/

public class Find_All_Mincuts extends Algorithm

{

    //This Ke will be used to store temp ke values for the graphs that are obtained by subtracting //every Ke combination of edges of the main graph

    int tempKe = 0;

    int Ke = 0; //this Ke will be used to store ke value of the main graph

    //Indicate the edges does not containes non-zero flow.

    public static final Color OTHER=Edge.untouchedColor;

    //Indicate the edges containes non-zero flow.

    public static final Color FLOW_EDGE=Edge.normalColor;

    //The residual graph of  graph

    private Graph resG;

    //Vector to store identities of edges that are in mincut sets

    protected Vector Critical = new Vector();

    //Constructor of algorithm Find_All_Mincuts.
```

```
public Find_All_Mincuts()
{
	legendPrompt=new String[]{"Non-Critical","Critical"};
	legendColor=new Color[]{OTHER,FLOW_EDGE};
	resG = new Graph();
} //End of Constructor
/**
*Set arguments from the argument list.
*Valid args: null, Object[]{Integer(int sourceNode) , *Integer(int *sinkNode)}
*@param args The list of arguments.
*@exception IllegalArgumentException Illegal argument defined.
*/
protected void setArg(Object args[]) throws IllegalArgumentException
{
	Enumeration enum=graph.verticesElements();
	Vertex v=(Vertex) enum.nextElement();
	startNode=v.getID();
	if (!enum.hasMoreElements())
	{
		throw new InvalidGraphTypeException("Graph must contain more than 2 vertices.");
	}
	v=(Vertex) enum.nextElement();
	endNode=v.getID();
	if (args==null)
		return;
	if (args.length!=2)
		throw new IllegalArgumentException("Wrong number of arguments");
	switch(args.length)
	{
		case 2:
		try
		{
			startNode=((Integer) args[0]).intValue();
		}catch(Exception e)
		{
			throw new IllegalArgumentException("Wrong format of arguments[0]");
		}
		try
		{
			endNode=((Integer) args[1]).intValue();
		}catch(Exception e)
		{
			throw new IllegalArgumentException("Wrong format of arguments[1]");
```

```
            }
            break;
        } //End switch
}// End setArg(Object args[])
/**
*Preferred dialog: Ask2VertexDialog.
*@param graph The Graph.
*@param fr The parent Frame.
*@return the preferred dialog.
*/
public GenericDialog getPreferredDialog(Graph graph,Frame fr)
{
        return new Ask2VertexDialog(graph,fr,"Vertex Dialog","Input the source vertex and sink vertex?",true);
} //End getPreferredDialog(Graph graph,Frame fr)

private boolean augmentingPathExist(Graph graph)
{
        resG=(Graph) graph.clone();
        ResidualGraph rg=new ResidualGraph();
        rg.execute(resG,new Object[]{new Integer(startNode),new Integer(endNode)});
        BFS bfs=new BFS();
        bfs.execute(resG,new Object[]{new Integer(startNode),new Boolean(false)});
        if (resG.getPredecessorNode(endNode)==resG.NUL)
        {
                return false;
        }
        return true;
} // End augmentingPathExist(Graph graph)

private void addAugmenting(Graph graph)
{
        double cf=Double.POSITIVE_INFINITY;
        double d;
        int u,v;
        v=endNode;
        for(u=resG.getPredecessorNode(v);u!=Graph.NUL;u=resG.getPredecessorNode(v))
        {
                d=graph.getEdge(u,v).getWeight()-graph.getEdge(u,v).getFlow();
                if (cf>d)
                        cf=d;
                v=u;
        } //End for
        v=endNode;
```

```
        for(u=resG.getPredecessorNode(v);u!=Graph.NUL;u=resG.getPredecessorNode(v))
        {
                graph.getEdge(u,v).setFlow(graph.getEdge(u,v).getFlow()+cf);
                 v=u;
        }//End for
} //End addAugmenting(Graph graph)

private void colorFlow(Graph graph)
{
        for(Enumeration e=graph.allEdgesElements();e.hasMoreElements();)
        {
                Edge edge=(Edge) e.nextElement();
                if(edge.critical)
                {
                        paintCritical(edge);
                } else
                {
                        paintNonCritical(edge);
                }
        } //end for
        graph.getVertex(startNode).setColor(Color.green);
        graph.getVertex(endNode).setColor(Color.yellow);
} //end of colorFlow()

private double flowOut(Graph graph,int node)
{
        Edge edge;
        double flow=0.0;
        for(Enumeration e=graph.edgesElements(node);e.hasMoreElements();)
        {
                edge=(Edge) e.nextElement();
                flow+=edge.getFlow();
        }
        return flow;
} //End flowOut(Graph graph,int node)
/**
*Implementation of this algorithm.
*@return <code> Double(double <I> flow_of_the_graph</I>) </code>
*/
protected Object algorImpl()
{
        Edge edge;
        double flow;
```

```
        //we don't need to show flows
        graph.setShowFlow(false);
        //initially set all flows to zero
        for(Enumeration e=graph.allEdgesElements();e.hasMoreElements();)
        {
                edge=(Edge) e.nextElement();
                edge.setFlow(0.0);
        }
        while(augmentingPathExist(graph))
        {
                addAugmenting(graph);
        }
        flow=flowOut(graph,startNode);
        //Since edges have unit capacity of 1, the value of flow gives us Ke
        Ke = (int)flow;

        //Write the Ke value
        outputMessage="Ke = "+StringTool.doubleToString(flow,2) + "    ";
        //This function will iterate deleting every possible Ke combination and computing Ke to //identify Mincuts
        iterate(graph);
        return new Double(flow);
} //end of algorImpl()
//this function sets the tempKe. this value will be used to store Ke values of the graphs //obtained on each iteration
protected void setKe (int ke)
{
        tempKe = ke;
}

//function to color critical edges to red
protected void paintCritical(Edge e)
{
        Edge edge = e;
        edge.setColor(Color.red);
}
//function to color noncritical edges to blue
protected void paintNonCritical(Edge e)
{
        Edge edge = e;
        edge.setColor(Color.blue);
}
/**
*This function determines the identities of critical edeges based on the criteria of computed *Ke of two nodes on a given graph. It tests each and every possible Ke combination of edges *of the graph.
```

```
*/
protected void iterate (Graph graph)
{
	Graph ana = graph;
	Graph tempGraph = ana;
	ana.enumerateEdges();
	int from = 0;
	int to = 0;
	int numOfEdges = ana.getNumOfEdges();
	Edge temp1 = null; //temp edges to store deleted edges so that we can add them back
	Edge temp2 = null;
	Edge temp3 = null;
	Edge temp4 = null;
	Edge edge = null;
	//Create an instance of textArea to display the results
	Show_All_Mincuts criticalEdges = new Show_All_Mincuts();
	//We need Ke nested for loops to test all Ke combination of edges
	switch (Ke)
	{
	case 1:
	//we need ke times nested for loops to try every combination of ke edge pairs
	for(int i = 0 ; i < numOfEdges ; i++)
	{
		//get an edge
		temp1 = tempGraph.getEdge(i);
		//delete this edge from the graph
		tempGraph.deleteEdge(temp1);
		//find Ke on this new graph
		tempKe = (int)reFindKe(tempGraph);
		//test if Ke has dropped to zero which means this edge is critical, add it to Vector //Critical
		if (tempKe == 0)
		{
			Critical.add(temp1);
		} //end if
		//Add the deleted edge back for next iteration, make sure that it has the same //edgeId //as before
		tempGraph.addEdge(temp1, temp1.getEdgeID());
	} //end for
	break; //end of case 1
	case 2:
	//we need ke times nested for loops to try every combination of ke edge pairs
	for(int i = 0 ; i < numOfEdges ; i++)
	{
		for (int j = i+1; j < numOfEdges; j++)
```

```
        {
            //get a pair of edges
            temp1 = tempGraph.getEdge(i);
            temp2 = tempGraph.getEdge(j);
            //delete this pair of edges from the graph
            tempGraph.deleteEdge(temp1);
            tempGraph.deleteEdge(temp2);
            //find Ke on this new graph
            tempKe = (int)reFindKe(tempGraph);
            //test if Ke has dropped to zero which means this pair is critical, add them to //Vector Critical
            if (tempKe == 0)
            {
                Critical.add(temp1);
                Critical.add(temp2);
            } //end if
            //Add the deleted edges back for next iteration, make sure that they have the //same edgeIds as
            before
            tempGraph.addEdge(temp1, temp1.getEdgeID());
            tempGraph.addEdge(temp2, temp2.getEdgeID());
        } //end for
    } //end for
    break; //end of case 2
    case 3:
    //we need ke times nested for loops to try every combination of ke edge pairs
    for(int i = 0 ; i < numOfEdges ; i++)
    {
        for (int j = i+1; j < numOfEdges; j++)
        {
            for(int k = j+1; k < numOfEdges; k++)
            {
                //get the a pair of edges
                temp1 = tempGraph.getEdge(i);
                temp2 = tempGraph.getEdge(j);
                temp3 = tempGraph.getEdge(k);
                //delete this pair of edges from the graph
                tempGraph.deleteEdge(temp1);
                tempGraph.deleteEdge(temp2);
                tempGraph.deleteEdge(temp3);
                //find Ke on this new graph
                tempKe = (int)reFindKe(tempGraph);
                //test if Ke has dropped to zero which means this pair is critical
                if (tempKe == 0)
                {
```

```
                    Critical.add(temp1);
                    Critical.add(temp2);
                    Critical.add(temp3);
                } //end if
                //Add the deleted edges back for next iteration, make sure that they have the //same edgeIds as before
                tempGraph.addEdge(temp1, temp1.getEdgeID());
                tempGraph.addEdge(temp2, temp2.getEdgeID());
                tempGraph.addEdge(temp3, temp3.getEdgeID());
            } //end for
        } //end for
    } //end for
    break; //end of case 3
    case 4:
    //we need ke times nested for loops to try every combination of ke edge pairs
    for(int i = 0 ; i < numOfEdges ; i++)
    {
        for (int j = i+1; j < numOfEdges; j++)
        {
            for(int k = j+1; k < numOfEdges; k++)
            {
                for (int n = k+1; n < numOfEdges; n++)
                {
                    //get the a pair of edges
                    temp1 = tempGraph.getEdge(i);
                    temp2 = tempGraph.getEdge(j);
                    temp3 = tempGraph.getEdge(k);
                    temp4 = tempGraph.getEdge(n);
                    //delete this pair of edges from the graph
                    tempGraph.deleteEdge(temp1);
                    tempGraph.deleteEdge(temp2);
                    tempGraph.deleteEdge(temp3);
                    tempGraph.deleteEdge(temp4);
                    //find Ke on this new graph
                    tempKe = (int)reFindKe(tempGraph);
                    //test if Ke has dropped to zero which means this pair is critical
                    if (tempKe == 0)
                    {
                        Critical.add(temp1);
                        Critical.add(temp2);
                        Critical.add(temp3);
                        Critical.add(temp4);
                    } //end if
```

```
                        //Add the deleted edges back for next iteration, make sure that they //have the same edgeIds as before
                        tempGraph.addEdge(temp1, temp1.getEdgeID());
                        tempGraph.addEdge(temp2, temp2.getEdgeID());
                        tempGraph.addEdge(temp3, temp3.getEdgeID());
                        tempGraph.addEdge(temp4, temp4.getEdgeID());
                    } //end for
                } //end for
            } //end for
        } //end for
        break; //end of case 4
        default:
        System.out.println("Ke is too big, no switch case for this Ke");
        criticalEdges.setText("Ke is too big, no switch case for this Ke");
        } //end switch
        //Array to identify the critical edges
        int [] criticalEdgeIdentities = new int [numOfEdges];
        //Initialize the array
        for (int m = 0 ; m < criticalEdgeIdentities.length ; m++)
        {
            criticalEdgeIdentities [m] = 0;
        }
        //Print out the critical edge pairs
        System.out.println("Mincut sets are : ");
        if (Ke == 1)
        {
            criticalEdges.appendText("There are " + (Critical.size()) + " Mincut sets : ");
            for (int k = 0 ; k < Critical.size(); k++)
            {
                Edge e1 = (Edge)Critical.get(k);
                criticalEdgeIdentities [e1.getEdgeID()] = 1;
                ana.getEdge(e1.getEdgeID()).setCritical(true);
                System.out.println(e1.getEdgeID() + "  ");
                criticalEdges.appendText(e1.getEdgeID() + "  ");
            } //end for
        } //end if
        if (Ke == 2)
        {
            criticalEdges.appendText("There are " + (Critical.size())/2 + " Mincut sets : ");
            for (int k = 0 ; k < Critical.size(); k++)
              {
                Edge e1 = (Edge)Critical.get(k);
                Edge e2 = (Edge)Critical.get(k+1);
```

```
                criticalEdgeIdentities [e1.getEdgeID()] = 1;
                criticalEdgeIdentities [e2.getEdgeID()] = 1;
                System.out.println("(" + e1.getEdgeID() + "," + e2.getEdgeID() + ")");
                criticalEdges.appendText(" (" + e1.getEdgeID() + " , " + e2.getEdgeID() + ") ");
                k= k+1;
        } //end for
} //end if
if (Ke == 3)
{
        criticalEdges.appendText("There are " + (Critical.size())/3 + " Mincut sets : ");
        for (int k = 0 ; k < Critical.size(); k++)
        {
                Edge e1 = (Edge)Critical.get(k);
                Edge e2 = (Edge)Critical.get(k+1);
                Edge e3 = (Edge)Critical.get(k+2);
                criticalEdgeIdentities [e1.getEdgeID()] = 1;
                criticalEdgeIdentities [e2.getEdgeID()] = 1;
                criticalEdgeIdentities [e3.getEdgeID()] = 1;
                System.out.println("(" + e1.getEdgeID() + "," + e2.getEdgeID() + "," + e3.getEdgeID() + ")");
                criticalEdges.appendText( " (" + e1.getEdgeID() + " , " + e2.getEdgeID() + " , " + e3.getEdgeID() +
                ") ");
                k= k+2;
        } //end for
} //end if
if (Ke == 4)
{
        criticalEdges.appendText("There are " + (Critical.size())/4 + " Mincut sets : ");
        for (int k = 0 ; k < Critical.size(); k++)
        {
                Edge e1 = (Edge)Critical.get(k);
                Edge e2 = (Edge)Critical.get(k+1);
                Edge e3 = (Edge)Critical.get(k+2);
                Edge e4 = (Edge)Critical.get(k+3);
                criticalEdgeIdentities [e1.getEdgeID()] = 1;
                criticalEdgeIdentities [e2.getEdgeID()] = 1;
                criticalEdgeIdentities [e3.getEdgeID()] = 1;
                criticalEdgeIdentities [e4.getEdgeID()] = 1;
                System.out.println("(" + e1.getEdgeID() + "," + e2.getEdgeID() + "," + e3.getEdgeID() + "," +
                e4.getEdgeID() + ")");
                criticalEdges.appendText( " (" + e1.getEdgeID() + " , " + e2.getEdgeID() + " , " + e3.getEdgeID() +
                " , " + e4.getEdgeID() + ") ");
                k= k+3;
        } //end for
} //end if
```

```
        //identify the edges of Mincuts by printing out between which vertices they exist
        for (int k = 0 ; k < criticalEdgeIdentities.length; k++)
        {
            if (criticalEdgeIdentities [k] == 1)
            {
                Edge e1 =tempGraph.getEdge(k);
                System.out.println("Edge#" + e1.getEdgeID() + " is between vertex : " +
                e1.getFromNode() + " and vertex : " + e1.getToNode());
                criticalEdges.appendText("\nEdge#" + e1.getEdgeID() + " is between vertex : " + e1.getFromNode()
                + " and vertex : " + e1.getToNode());
             } //end if
        } //end for

        //Color edges of Mincuts to red only when Ke is equal to 1
        if(Ke == 1)
        {
            colorFlow(ana);
        }
        //identify edges with their edgeIds for visual clarity
        for (int d =0 ; d < numOfEdges ; d++)
        {
            ana.getEdge(d).setWeight(d);
        }
    } //end iterate()

    //This function computes Ke on the passed in graph,, and returns that value
    public double reFindKe(Graph g)
    {
        Graph changed = g;
        Edge edge;
        double flow = 0;

        for(Enumeration e=changed.allEdgesElements();e.hasMoreElements();)
        {
            edge=(Edge) e.nextElement();
            edge.setFlow(0.0);
        }

        while(augmentingPathExist(changed))
        {
            addAugmenting(changed);
        }
        flow=flowOut(changed,startNode);
```

```
            return flow;
        } //end of reFindKe()
} //end of Find_All_Mincuts()
```

2. Find_Ke.java

```
package tw.edu.ncnu.im.cnclab.Algor;
import tw.edu.ncnu.im.cnclab.UI.*;
import tw.edu.ncnu.im.cnclab.DataStru.*;
import tw.edu.ncnu.im.cnclab.Tools.*;
import tw.edu.ncnu.im.cnclab.JGAP.*;
import tw.edu.ncnu.im.cnclab.JGAP.UI.*;
import java.util.*;
import java.awt.*;
/**
*Find_Ke class is a modified version of Ford-Fulkerson maximum flow algorithm that is included *in original JGAP platform This class finds the maximum number of edge-disjoint paths between *a source and a sink node on an undirected graph.Algorithm achieves this by assigning a unit flow *capacity of 1 to every edge of the graph and running Ford-Fulkerson maximum flow algorithm *on this graph.Disjoint paths that are found is painted with different colors to make them *visiulized more easily. These paths are also identified in a seperate text box by their vertex *squence from source to destination. One important thing to notice, the disjoint paths found by *this algorithm are the shortest ones. There may be alternative disjoint paths but they are not listed *since there is a shorter disjoint path than that.
*@Author Baris AKTOP
*Modified for finding maximum number of edge disjoint paths betwwen two nodes
*of an arbitrary undirected graph
*March 2003
*/
public class Find_Ke extends Algorithm
{
    protected int verts;
    protected int Ke;
    //Indicate the edges does not containes non-zero flow.
    public static final Color OTHER=Color.gray;
    //Indicate the edges containes non-zero flow.
    public static final Color FLOW_EDGE=Edge.normalColor;
    //The residual graph of <code> graph </code>
    private Graph resG;
    //Constructor of algorithm Ford-Fulkerson.
    public Find_Ke()
    {
        legendPrompt=new String[]{"First Path","Second Path","Third Path","Fourth Path", "Fifth Path","Sixth Path", "Other"};
        legendColor=new Color[]{Color.blue, Color.green, Color.red, Color.cyan, Color.orange, Color.white, Color.gray};
        resG=new Graph();
    } //End Constructor
```

```
/**
*Set arguments from the argument list.
*Valid args: null, Object[] {Integer (int  sourceNode ) , Integer(int sinkNode)}
*@param args The list of arguments.
*@exception IllegalArgumentException Illegal argument defined.
*/
protected void setArg(Object args[]) throws IllegalArgumentException
{
	Enumeration enum=graph.verticesElements();
	Vertex v=(Vertex) enum.nextElement();
	startNode=v.getID();
	if (!enum.hasMoreElements())
	{
		throw new InvalidGraphTypeException("Graph must contain more than 2 vertices.");
	}
	v=(Vertex) enum.nextElement();
	endNode=v.getID();
	if (args==null)
		return;
	if (args.length!=2)
		throw new IllegalArgumentException("Wrong number of arguments");
	switch(args.length)
	{
		case 2:
		try
		{
			startNode=((Integer) args[0]).intValue();
		}catch(Exception e)
		{
			throw new IllegalArgumentException("Wrong format of arguments[0]");
		}
		try
		{
			endNode=((Integer) args[1]).intValue();
		}catch(Exception e)
		{
			throw new IllegalArgumentException("Wrong format of arguments[1]");
		}
		break;
	} //end switch
} //end setArg(Object args[])
/**
*Preferred dialog: Ask2VertexDialog.
```

```
*@param graph The Graph.
*@param fr The parent Frame.
*@return the preferred dialog.
*/
public GenericDialog getPreferredDialog(Graph graph,Frame fr)
{
	return new Ask2VertexDialog(graph,fr,"Vertex Dialog","Input the source vertex and sink vertex?",true);
}

private boolean augmentingPathExist(Graph graph)
{
	resG=(Graph) graph.clone();
	ResidualGraph rg=new ResidualGraph();
	rg.execute(resG,new Object[]{new Integer(startNode),new Integer(endNode)});
	BFS bfs=new BFS();
	bfs.execute(resG,new Object[]{new Integer(startNode),new Boolean(false)});
	if (resG.getPredecessorNode(endNode)==resG.NUL)
	{
		return false;
	}
	return true;
} //end augmentingPathExist(Graph graph)
Stack path = new Stack(); //stack to store path IDs

private void addAugmenting(Graph graph)
{
	double cf=Double.POSITIVE_INFINITY;
	double d;
	int u,v;
	v=endNode;
	for(u=resG.getPredecessorNode(v);u!=Graph.NUL;u=resG.getPredecessorNode(v))
	{
		d=graph.getEdge(u,v).getWeight()-graph.getEdge(u,v).getFlow();
		if (cf>d)
		cf=d;
		v=u;
	} //end for
	v=endNode;
	for(u=resG.getPredecessorNode(v);u!=Graph.NUL; u=resG.getPredecessorNode(v))
	{
		graph.getEdge(u,v).setFlow(graph.getEdge(u,v).getFlow()+cf);
		//push nodes into a stack starting from the sink node so that
		//we can print the paths after all the computation is done
```

```
                path.push(graph.getVertex(v));
                v=u;
        } //end for
        //add the source node lastly
        path.push(graph.getVertex(v));
} //end addAugmenting(Graph graph)

int i = 0; //flag to reset the graph only for the first time
private void colorFlow(Graph graph , Color color, int pathId)
{
        Color newcolor = color;
        if(i==0)
        {
        graph.reset();
        }
        i += 1; //set flag to prevent from reseting the graph on further coloring of other dijoint //paths
        for(Enumeration e=graph.allEdgesElements();e.hasMoreElements();)
        {
                Edge edge=(Edge) e.nextElement();
                if (edge.getPathID() == pathId )
                {
                        edge.setColor(newcolor);
                }else if(i == (Ke) && edge.getFlow() == 0.0)
                {
                        edge.setColor(OTHER);
                }
        } //end for
        graph.getVertex(startNode).setColor(Color.green);
        graph.getVertex(endNode).setColor(Color.yellow);
} //end of colorFlow(Graph graph , Color color, int pathId)

private double flowOut(Graph graph,int node)
{
        Edge edge;
        double flow=0.0;
        for(Enumeration e=graph.edgesElements(node);e.hasMoreElements();)
        {
                edge=(Edge) e.nextElement();
                flow+=edge.getFlow();
        }
        return flow;
} //end flowOut(Graph graph,int node)
```

```
protected void setKe (double ke)
{
	Ke = (int) ke;
}

public int getKe ()
{
	return Ke;
}
/*
*This function labels edges with appropriate pathId attributes
*@param vector The vector that holds the vertices on this disjoint path
*@param id The pathId of this disjoint path that will be assigned to its edges
*@return void
*/
protected void edgeIdentifier(Vector v , int id)
{
	Edge e;
	Vector myVector = v;
	//Since our vector holds the vertices of this dijoint path sequentially we can easily //identify edges on this path
	and set its pathID attribute This attribute will help us to color //each disjoint path with a different color
	for (int i = 0; i < myVector.size()-1; i++)
	{
		e=graph.getEdge(((Vertex)myVector.get(i)).getID(), ((Vertex)myVector.get(i+1)).getID());
		e.setPathID(id);
	} //end for
} //end of method edgeIdentifier()
/**
*Implementation of this algorithm.
*@return <code> Double(double <I> flow_of_the_graph</I>) </code>
*/
protected Object algorImpl()
{
	Vertex v;
	Edge edge;
	int from, to, node;
	int counter=0;
	double flow;
	graph.setShowFlow(true);
	for(Enumeration e=graph.allEdgesElements();e.hasMoreElements();)
	{
		edge=(Edge) e.nextElement();
		edge.setFlow(0.0);
```

```
} //end for

while(augmentingPathExist(graph))

{

        addAugmenting(graph);

} //end while

//since all of the edge capacities equal to 1 total flow gives us the number of edge disjoint //paths between source and sink

flow=flowOut(graph,startNode);

setKe(flow);

outputMessage="Ke = "+StringTool.doubleToString(flow,2) + "    ";

//print out the sequence of the vertices on disjoint paths found between source and sink //node. There will be ke=flow disjoint paths to be printed

ShowPathsTextArea disjointPaths = new ShowPathsTextArea();

if((int)flow == 0)

{

        System.out.println("THERE ARE NO DISJOINT PATHS BETWEEN NODE ID= "

        + startNode + " AND NODE ID= " + endNode);

        disjointPaths.setText("THERE ARE NO DISJOINT PATHS BETWEEN NODE ID= " + startNode + "
        AND NODE ID= " + endNode);

}else

{

        System.out.println("THERE ARE " + (int)flow + " DISJOINT PATHS BETWEEN NODE ID= " +
startNode + " AND NODE ID= " + endNode);

disjointPaths.setText("THERE ARE " + (int)flow + " DISJOINT PATHS BETWEEN NODE ID= " +
startNode + " AND NODE ID= " + endNode);

} //end else-if

//Array of Vectors to store the vertices of each disjoint path

Vector [] routes = new Vector [Ke];

//Array of Colors to help painting each disjoint path with a different color

Color [] paint = {Color.blue, Color.green, Color.red, Color.cyan, Color.orange, Color.white};

Color color;

Ke = (int)flow;

for (int k = 1; k <= Ke; k++)

{

        System.out.println("path#" + k + " =");

        disjointPaths.appendText("\npath#" + k + " =");

        routes [k-1] = new Vector();

        while(!path.empty())

        {

                //pop from the stack and make sure its casted to Vertex

                Vertex id =(Vertex)path.pop();

                int vertex = id.getID(); //get the id of the vertex

                (routes [k-1]).add(id); //add vertices to a vector of array routes

                System.out.println(vertex); //start printing vertex that is popped out

                disjointPaths.appendText(" " + vertex + ",");
```

```
                    if (vertex == endNode) //if we reach the endNode then this path is finished
                        break;
                } //end while
                //call this function to label edges with appropriate pathID attributes
                edgeIdentifier(routes [k-1], k);
                color = paint [(k-1)%6]; //Pick a different color at each time
                colorFlow(graph,color,k); //paint this path
            } //end for
            return new Double(flow);
        } //end of algorImpl()
} //end of Find_Ke()
```

3. Show_All_Mincuts.java

```
package tw.edu.ncnu.im.cnclab.UI;
import java.awt.*;
import java.awt.event.*;
/**
*Description: This class shows all Mincuts in a separate window
*@author  Baris AKTOP
*/
public class Show_All_Mincuts extends Frame implements ActionListener
{
    TextArea showCrtEdges = new TextArea();
    Button close = new Button("CLOSE");
    Font f1 = new Font (" ", Font.BOLD, 14);
    public Show_All_Mincuts()
    {
        super ("ALL MINCUTS");
        showCrtEdges.setEditable(false);
        showCrtEdges.setFont(f1);
        close.setFont(f1);
        add (showCrtEdges, BorderLayout.CENTER);
        add (close, BorderLayout.SOUTH);
        close.setBackground(Color.red);
        close.addActionListener(this);
        close.setSize(50,20);
        setSize(525,200);
        clearText ();
        show();
    } //end of constructor

    public void setText(String msg)
```

```
        {
            this.showCrtEdges.setText(msg);
        }

        public void appendText(String msg)
        {
            this.showCrtEdges.append(msg);
        }

        public void clearText ()
        {
            this.showCrtEdges.setText("");
        }

        public void actionPerformed(ActionEvent e)
        {
            this.dispose();
        }

        public static void main(String [] args)
        {
            new Show_All_Mincuts();
        }
} //end of class ShowCriticalEdges()
```

4. ShowPathsTextArea.java

```
package tw.edu.ncnu.im.cnclab.UI;
import java.awt.*;
import java.awt.event.*;
/**
*Description: This class shows all disjoint paths in a separate window
*@author  Baris AKTOP
*/
public class ShowPathsTextArea extends Frame implements ActionListener
{
    TextArea showPaths = new TextArea();
    Button close = new Button("CLOSE");
    Font f1 = new Font (" ", Font.BOLD, 14);
    public ShowPathsTextArea()
    {
        super ("SHORTEST DISJOINT PATHS");
        showPaths.setEditable(false);
```

```
        showPaths.setFont(f1);
        close.setFont(f1);
        add (showPaths, BorderLayout.CENTER);
        add (close, BorderLayout.SOUTH);
        close.setBackground(Color.red);
        close.addActionListener(this);
        close.setSize(50,20);
        setSize(525,200);
        clearText ();
        show();
    } //end of constructor
    public void setText(String msg)
    {
        this.showPaths.setText(msg);
    }

    public void appendText(String msg)
    {
        this.showPaths.append(msg);
    }

    public void clearText ()
    {
        this.showPaths.setText("");
    }

    public void actionPerformed(ActionEvent e)
    {
        this.dispose();
    }

    public static void main(String [] args)
    {
        new ShowPathsTextArea();
    }
} //end of class ShowPathsLabel()
```

B. MODIFIED CLASSES OF JGAP

1. RandomDialog.java

```
package tw.edu.ncnu.im.cnclab.JGAP.UI;
import tw.edu.ncnu.im.cnclab.UI.*;
import tw.edu.ncnu.im.cnclab.JGAP.*;
```

```
import java.awt.*;
import java.awt.event.*;
/**
*The  RandomDialog class show the dialog of random graph.
*@see tw.edu.ncnu.im.cnclab.JGAP.GraphRandomizer
*/
class RandomDialog extends GenericDialog implements ActionListener
{
	Panel defPanel;
	Panel boundPanel;
	Panel def2Panel;
	//TextField for Vertices.
	TextField verticesField;
	//TextField for density.
	TextField densityField;
	//TextField for edgeWeight.
	TextField edgeWeightField;
	int vertices;
	double density;
	/**
	*Construct a RandomDialog.
	*@param fr  the parent window.
	*@param title the window title.
	*@param promptText the prompt text. '\n' can be used as separator.
	*@param isModal whether the dialog modal.
	*/
	public RandomDialog(Frame fr, String title, String promptText,boolean isModal)
	{
		super(fr, title, promptText, isModal,40);
	}
	/**
	*Initite the custom panel(central)Panel.
	*@see tw.edu.ncnu.im.cnclab.UI.GenericDialog#centralPanel
	*/
	protected void setCentralPanel()
	{
		centralPanel.setLayout(new BorderLayout());
		defPanel=new Panel();
		setDefPanel();
		centralPanel.add("North",defPanel);
		boundPanel=new Panel();
		setBoundPanel();
		centralPanel.add("Center",boundPanel);
```

```
        def2Panel=new Panel();
        centralPanel.add("South",def2Panel);
} //end of setCentralPanel()

void setDefPanel()
{
        defPanel.add(new Label("Vertices [1-"+Graph.MAX_VERTICES+"]:"));
        verticesField=new TextField(1);
        defPanel.add(verticesField);
        verticesField.setText("5");
        defPanel.add(new Label("   ke:[1-6]"));
        densityField=new TextField(1);
        defPanel.add(densityField);
        densityField.setText("2");
} //end of setDefPanel()

void setBoundPanel()
{
        boundPanel.add(new Label("Unit Edge Weight:"));
        edgeWeightField=new TextField(1);
        boundPanel.add(edgeWeightField);
        edgeWeightField.setText("1.0");
        edgeWeightField.setEditable(false);
} // end of setBoundPanel()
/**
*Check if the input format valid.
*@return <code> true </code> if inputs is valid; <code> false </code> if otherwise.
*/
protected boolean checkFormat()
{
        Integer I;
        Double D;
        if ((I=getIntegerFieldValue(verticesField,"Vertices",1,Graph.MAX_VERTICES))==null)
        {
                return false;
        }//end if
        vertices=I.intValue();
        if ((D=getDoubleFieldValue(densityField,"Ke",1.0,6.0))==null)
        {
                return false;
        }//end if
        density=D.doubleValue();
        //ke cannot be >= number of vertices
```

```
        if ((int)getDensity() >= getVertices())
        {
                msgField.setText("ke cannot be greater than or equal to verts");
                return false;
        } //end if
        return true;
} //end checkFormat()
/**
*Get the number of vertices.
*@return the number of vertices.
*/
public int getVertices()
{
        return vertices;
}
/**
*Get the probability of edge appear between vertices.
*@return the probability of edge appear between vertices.
*/
public double getDensity()
{
        return density;
}
/**
*Return the parameter that the dialog get.
*@param additionalArg Additional arguments for another method.
*@return The parameters of the dialog get. (Default: additionalArg
*Array[0]:  the Integer(vertices)
*<ui> Array[1]:  the Double(density)
*/
public Object[] getArg(Object[] additionalArg)
{
        Object[] oArray;
        if (additionalArg!=null)
        {
                oArray=new Object[additionalArg.length+2];
                for(int i=0;i<additionalArg.length;i++)
                {
                        oArray[i+2]=additionalArg[i];
                }
        } else
        {
                oArray=new Object[2];
```

```
            } //end if-else
            oArray[0]=new Integer(vertices);
            oArray[1]=new Double(density);
            return oArray;
        } //end of Object[] getArg(Object[] additionalArg)
} //end of RandomDialog
```

2. GraphRandomizer.java

```
//Modified by Baris AKTOP
package tw.edu.ncnu.im.cnclab.JGAP;
import tw.edu.ncnu.im.cnclab.DataStru.*;
import java.io.Serializable;
import java.util.*;
import java.awt.*;

public class GraphRandomizer
{
    //The Random.
    protected Random r;
    //Indicated the width of canvas.
    protected int canvasWidth;
    //Indicated the height of canvas.
    protected int canvasHeight;
    //Constructor of GraphRandomizer
    public GraphRandomizer(Dimension d)
    {
        r=new Random();
        canvasWidth=d.width;
        canvasHeight=d.height;
    } //end of constructor
    /**
    *Generate a randomized graph.
    *@param verts  the number of vertices of the graph.
    *@param ke  minimum edge connectivity between vertices
    *@return the generated graph.
    */
    public Graph generate(int verts,double ke)
    {
        Graph g=new Graph();
        if (verts>g.MAX_VERTICES)
            verts=g.MAX_VERTICES;
        generateVertices(g,verts);
```

```
        generateEdges(g,verts,ke);
        g.setDirected(false);
        return g;
} //end of generate(int verts,double ke)
/**
*Randomly place the vertices.
*@param graph  the graph
*@param verts  the number of vertices of the graph.
*/
protected  void generateVertices(Graph graph,int verts)
{
        int x,y;
        for(int i=0;i<verts;i++)
        {
                do{
                        x=Math.abs(r.nextInt())%(canvasWidth-graph.vertexDiameter) +graph.vertexDiameter/2;
                        y=Math.abs(r.nextInt())%(canvasHeight-graph.vertexDiameter) +graph.vertexDiameter/2;
                        }while(graph.closeTo(x,y)>=0);
                graph.makeVertex(x,y);
        } //end for
} // end of generateVertices(Graph graph,int verts)
/**
*Randomly generate the edge.
*@param graph  the graph
*@param verts  the number of vertices of the graph.
*@param dke  minimum connectivity between vertices(double)
*/
protected void generateEdges(Graph graph,int verts,double dke)
{
        //define the minimum disjoint-edge connectivity of the graph
        int ke = (int)dke ;
        //ke cannot be >= verts
        if (ke < verts)
        {
                //define an array to store degrees of vertices
                int [] used;
                used = new int [verts];
                //initialize the array used
                for (int k=0; k < used.length; k++)
                {
                        used[k] = 0;
                } // end for
                int j = 0;
```

```
                for(int i=0;i<verts;i++)
                {
                    if (used [i] < ke)
                    {
                        do
                        {
                            //the graph is not self looped, so i cannot be equal to j
                            do
                            {
                                j = r.nextInt(verts);
                            }while ((i == j) || graph.isEdge(i,j) || used [j] > ke);
                            //create unit weight and zero flow edge
                            graph.addEdge(i,j,1,0.0);
                            //increase the degree of this vertices pair
                            used [i] += 1;
                            used [j] += 1;
                        }while ((used [i] < ke));
                    } //end if
                } // end of for
            } //end if
        } //end of method generateEdges()
} //end of class GraphRandomizer
```

3. AlgorithmMenu.java

```
package tw.edu.ncnu.im.cnclab.JGAP.UI;
import tw.edu.ncnu.im.cnclab.UI.*;
import tw.edu.ncnu.im.cnclab.JGAP.*;
import tw.edu.ncnu.im.cnclab.DataStru.*;
import tw.edu.ncnu.im.cnclab.Algor.*;
import tw.edu.ncnu.im.cnclab.Algor.UI.*;
import tw.edu.ncnu.im.cnclab.Algor.Util.*;
import tw.edu.ncnu.im.cnclab.Tools.*;
import java.awt.*;
import java.awt.event.*;
import java.util.EventObject;
/**
*The AlgorithmMenu defines the menu Algorithm.
*Developer may add graph algorithms MenuItem by modifing this class.
*Modified by Baris AKTOP
*/
class AlgorithmMenu
{
```

```
// private Menu algorithmMenu;
private Menu algorithmMenu;
private MenuItem Find_KeMenuItem;
private MenuItem Find_All_MincutsMenuItem;
private JGAPFrame parent;
private Ask2VertexDialog ask2VertexDialog;

AlgorithmMenu(JGAPFrame parent,Menu algorithmMenu)
{
    this.parent=parent;
    this.algorithmMenu=algorithmMenu;
    algorithmMenu();
} //end of AlgorithmMenu(JGAPFrame parent,Menu algorithmMenu)

private void algorithmMenu()
{
    Find_All_MincutsMenuItem = new MenuItem("Find All Mincuts");
    algorithmMenu.add(Find_All_MincutsMenuItem);
    Find_All_MincutsMenuItem.addActionListener(parent);
    Find_KeMenuItem = new MenuItem("Find Ke");
    algorithmMenu.add(Find_KeMenuItem);
    Find_KeMenuItem.addActionListener(parent);
} //end of algorithmMenu()

boolean menuActionPerformed(ActionEvent actionEvent)
{
    Object source=actionEvent.getSource();
    int dialogReturn;
    if (source == Find_All_MincutsMenuItem)
    {
        Find_All_Mincuts crtEdges=new Find_All_Mincuts();
        if (ask2VertexDialog==null)
            ask2VertexDialog=new Ask2VertexDialog(parent.graph,parent,"Vertex Dialog" , "Input the source
            vertex and sink vertex.", true);
        ask2VertexDialog.show();
        dialogReturn=ask2VertexDialog.getDialogReturn();
        if (dialogReturn!=GenericDialog.Cancel)
        {
            if (parent.graph.isModified())
            {
                parent.jgapMenu.hm.setHistoryBefore("Find All Mincuts");
            } //end if
            crtEdges.show(parent,parent.graph,ask2VertexDialog.getArg(null));
```

```
                } //end if
                return true;
            }else if (source == Find_KeMenuItem)
            {
                Find_Ke ke=new Find_Ke();
                if (ask2VertexDialog==null)
                    ask2VertexDialog=new Ask2VertexDialog(parent.graph,parent,"Vertex Dialog" ,"Input the source vertex and sink vertex.",true);
                ask2VertexDialog.show();
                dialogReturn=ask2VertexDialog.getDialogReturn();
                if (dialogReturn!=GenericDialog.Cancel)
                {
                    if (parent.graph.isModified())
                    {
                        parent.jgapMenu.hm.setHistoryBefore("Find Ke");
                    }
                    ke.show(parent,parent.graph,ask2VertexDialog.getArg(null));
                } //end if
                return true;
            } //end else-if
            return false;
        } //end of menuActionPerformed(ActionEvent actionEvent)
} //end of AlgorithmMenu
```

4. Graph.java

```
package tw.edu.ncnu.im.cnclab.JGAP;
import tw.edu.ncnu.im.cnclab.JGAP.UI.*;
import tw.edu.ncnu.im.cnclab.DataStru.*;
import java.io.Serializable;
import java.util.*;
import java.awt.*;
/**
*The Graph class implements data structure of graphs.
*The implementation of the graph supports both adjacency matrix and adjacency
*list. Use enumerations as adjacency list; for loop as adjacency matrix.
*Node stands for identity of vertex
*@see java.util.Enumeration
*@see Vertex
*@see Edge
*/
public class Graph implements Serializable, Cloneable
{
```

```
static final long serialVersionUID = -7533844471607204671;
//Indicate null item
public static final int NUL = -1;
//This constant value indicates that the getting adjacency edges as out adjacency edges.
public static final int OUT_ADJACENCY=0;
//This constant value indicates that the getting adjacency edges as in adjacency edges.
public static final int IN_ADJACENCY=1;
//The serial number of the graph.
public int serialNo=0;
//The serial number of the vertex.
protected int vertexSerialNo;
//The serial number of the edge.
protected int edgeSerialNo;
//Indicate if the graph has been modified.
protected boolean modified=false;
//The ListArray that store vertices
protected ListArray vertexList;
//The ListArray that store edges
protected ListArray2 edgeList;
//Whether the graph is directed.
protected boolean directed;
//Whether the flow of the graph show at screen.
protected boolean showFlow;
//The maximum number of vertices capacity;
public static final int MAX_VERTICES=100;
//The diameter of a vertex;
public static int vertexDiameter = 22;
//The space between each two vertices
public static int vertexSpace = 2*vertexDiameter;
//The best fit canvas size of the graph
protected Dimension canvasSize;
//The description of the graph.
protected String description;
//Construct a directed graph.
public Graph()
{
    this(true);
}

/**Construct a graph.
*@param directed  whether the graph is directed.
*/
public Graph(boolean directed)
```

```
{
        this(directed,new Dimension(GraphCanvas.defaultWidth,GraphCanvas.defaultHeight));
}
/**
*Construct a graph.
*@param directed  whether the graph is directed.
*@param canvasSize the size of canvas.
*/
public Graph(boolean directed,Dimension canvasSize)
{
        serialNo=0;
        vertexSerialNo=0;
        edgeSerialNo=0;
        modified=false;
        showFlow=false;
        this.directed=directed;
        description=new String(JGAPFrame.emptyFileName);
        vertexList=new ListArray(MAX_VERTICES);
        edgeList=new ListArray2(MAX_VERTICES);
        this.canvasSize=new Dimension(canvasSize.width,canvasSize.height);
}
/**
*Get the description of the graph.
*@return the description of the graph.
*/
public String getDescr()
{
        return description;
}
/**
*Set the description of the graph.
*@param descr  The description of the graph.
*/
public void setDescr(String descr)
{
        description=descr;
}
/**
*Returns a deep-copied clone of this Graph.
*@return a deep-copied clone of this Graph.
*/
public Object clone()
{
```

```
        try
        {
                Graph g=(Graph)super.clone();
                g.copyFrom(this,true);
                return g;
        }catch (CloneNotSupportedException e)
        {
                throw new InternalError();
        }
} //end of clone()
/**
*Copies the components from specified Graph to this Graph
*@param  src the source Graph
*/
public void copyFrom(Graph src)
{
        copyFrom(src,false);
}

private void copyFrom(Graph src,boolean fromClone)
{
        Vertex v;
        Edge edge;
        int coord[];
        if (fromClone)
        {
                vertexList=new ListArray(MAX_VERTICES);
                edgeList=new ListArray2(MAX_VERTICES);
                description=new String(src.description);
                canvasSize=new Dimension(src.canvasSize.width,src.canvasSize.height);
        }else
        {
                clear();
                description=src.description;
                canvasSize.width=src.canvasSize.width;
                canvasSize.height=src.canvasSize.height;
        } //end else-if
        serialNo=src.serialNo;
        vertexSerialNo=src.vertexSerialNo;
        directed=src.directed;
        showFlow=src.showFlow;

        for(Enumeration e=src.vertexList.elements();e.hasMoreElements();)
```

```
        {
            v=(Vertex) e.nextElement();
            vertexList.addItemAt(new Vertex(),v.getID());
            ((Vertex)vertexList.itemAt(v.getID())).copyFrom(v);
        }

        for(Enumeration e=src.allEdgesElements();e.hasMoreElements();)
        {
            edge=(Edge) e.nextElement();
            addEdge(edge);
        }
        modified=src.modified;
    } // end of copyFrom(Graph src,boolean fromClone)
    /**
    *Reset special attribute of the graph.
    *@see Vertex#reset
    *@see Edge#reset
    *@see clean
    */
    public void reset()
    {
        for(Enumeration ev=vertexList.elements();ev.hasMoreElements();)
        {
            ((Vertex) ev.nextElement()).reset();
        }
        for(Enumeration ea=allEdgesElements();ea.hasMoreElements();)
        {
            ((Edge) ea.nextElement()).reset();
        }
    } // end of reset()
    /**
    *Reset all special attribute the graph.
    *@see Vertex#clean
    *@see Edge#clean
    *@see reset
    */
    public void clean()
    {
        cleanAllVertices();
        cleanAllEdges();
    }
    /**
    *Remove all vertices and edges.
```

```
*/
public void clear()
{
	clear(directed);
}
/**
*Remove all vertices and edges.
*@param directed  set the new graph type.
*/
public void clear(boolean directed)
{
	vertexList.deleteAllItems();
	edgeList.deleteAllItems();
	vertexSerialNo=0;
	modified=false;
	this.directed=directed;
}
/**
*Whether the graph is directed.
*@return whether the graph is directed.
*/
public boolean isDirected()
{
	return directed;
}
/**
*Specified whether the graph has been modified.
*@param modified   whether the graph has been modified.
*/
protected void setDirected(boolean directed)
{
	this.directed=directed;
}
/**
*Whether the graph has been modified.
*@return whether the graph has been modified.
*/
public boolean isModified()
{
	return modified;
}
/**
*Specified whether the graph has been modified.
```

```
*@param modified   whether the graph has been modified.
*/
public void setModified(boolean modified)
{
	this.modified=modified;
}
/**
*Whether the graph algorithms need to consider the flow of the graph.
*@return whether the graph algorithms need to consider the flow of the graph.
*/
public boolean isShowFlow()
{
	return showFlow;
}
/**
*Whether the graph algorithms need to consider the flow of the graph.
*@return whether the graph algorithms need to consider the flow of the graph.
*/
public void setShowFlow(boolean b)
{
	showFlow=b;
}
/**
*Symmetric Difference edges (Excludsive OR) with other graph.
*@param g the graph to process with this graph.
*/
public void symmetricDifference(Graph g)
{
	Vertex v;
	Edge edge;
	for(Enumeration ev=g.verticesElements();ev.hasMoreElements();)
	{
		v=(Vertex) ev.nextElement();
		if (!isVertexExist(v.getID()))
		{
			addVertex(v);
		}
	}
	for(Enumeration ee=g.allEdgesElements();ee.hasMoreElements();)
	{
		edge=(Edge) ee.nextElement();
		if (isEdge(edge.getFromNode(),edge.getToNode()))
		{
```

```
                deleteEdge(edge);
            }else
            {
                addEdge(edge);
            } // end if-else
        } //end for
} // end of symmetricDifference(Graph g)
/**
*Union edges with other graph.
*@param g the graph to union with this graph.
*/
public void union(Graph g)
{
        Vertex v;
        Edge edge;
        for(Enumeration ev=g.verticesElements();ev.hasMoreElements();)
        {
            v=(Vertex) ev.nextElement();
            if (!isVertexExist(v.getID()))
            {
                addVertex(v);
            } // end if
        } // end for
        for(Enumeration ee=g.allEdgesElements();ee.hasMoreElements();)
        {
            edge=(Edge) ee.nextElement();
            if (!isEdge(edge.getFromNode(),edge.getToNode()))
            {
                addEdge(edge);
            } // end if
        } //end for
} // end of union(Graph g)
/**
*Get the size of the canvas.
*@return the size of canvas
*/
public Dimension getCanvasSize()
{
        return canvasSize;
}
/**
*Set the size of the canvas.
*@param d The new size of canvas.
```

```
*/
public void setCanvasSize(Dimension d)
{
	canvasSize=d;
}

/**
*Add vertex at specified coordinate.
*@param x  the coordinate x of this vertex.
*@param y  the coordinate y of this vertex.
*/
public void addVertex(int x, int y)
{
	addVertex(x,y,Vertex.defaultColor);
}
/**
*Add vertex at specified coordinate.
*@param x  the coordinate x of this vertex.
*@param y  the coordinate y of this vertex.
*@param color  the color of this vertex.
*/
public void addVertex(int x, int y,Color color)
{
	modified=true;
	vertexList.addItemAt(new Vertex(vertexSerialNo,x,y,color),vertexSerialNo);
	vertexSerialNo++;
}
/**
*Add a clone of specified vertex into this graph.
*The method use the infomation of specified vertex to create another vertex.
*@param v  the vertex to add.
*/
public void addVertex(Vertex v)
{
	modified=true;
	vertexList.addItemAt(new Vertex(v.getID(),v.getX(),v.getY(),v.getColor()),v.getID());
	if (v.getID() >= vertexSerialNo)
	{
	vertexSerialNo=v.getID()+1;
	}
}
/**
*If there is a canvas space for vertex, then add vertex at specified coordinate.
```

```
*This method prevents vertices congesting in some place.
*If there is some vertices close to the coordinate of new vertex,
*makeVertex won't add vertex and return  NUL , otherwise
*add vertex in this graph and return the vertexID of the vertex to add.
*@param x  the coordinate x of this vertex.
*@param y  the coordinate y of this vertex.
*@return the vertex ID if sucess; NUL if fail.
*/
public int makeVertex(int x, int y)
{
	int k;
	if ((k=closeTo(x,y))!=NUL)
	{
		return NUL;
	}
	if (vertexSerialNo > MAX_VERTICES-1)
	{
		return NUL;
	}
	addVertex(x,y,Vertex.defaultColor);
	return vertexSerialNo - 1;
}
/**
*Return the node that contains the specified coordinate.
*@param x  the coordinate x.
*@param y  the coordinate y.
*@return The node that contains the specified coordinate; NUL if none of vertices contain the corrdinate.
*/
public int inAVertex(int i, int j)
{
	for (int k = 0; k < vertexSerialNo; k++)
	{
		if (!vertexList.itemExistAt(k))
			continue;
		if (Math.abs(((Vertex)vertexList.itemAt(k)).x - i) < vertexDiameter/2
		&& Math.abs(((Vertex)vertexList.itemAt(k)).y - j) < vertexDiameter/2)
			return k;
	} //end for
	return NUL;
} // end of inAVertex(int i, int j)
/**
*Return the node that close to specified coordinate.
*@param x  the coordinate x.
```

```
*@param y  the coordinate y.
*@return The node that close to specified coordinate; NUL if none of vertices close the corrdinate.
*/
public int closeTo(int x, int y)
{
	return closeTo(x,y,vertexSpace);
}
/**
*Return the node that close to specified coordinate.
*If Math.abs(vertex.x - x) < distance and  Math.abs(vertex.y)-y)<distance
*@param x  the coordinate x.
*@param y  the coordinate y.
*@param distance  the distance.
*@return The node that close to specified coordinate; NUL if none of vertices close the corrdinate.
*/
public int closeTo(int x, int y,int distance)
{
	for (int k = 0; k < vertexSerialNo; k++)
	{
		if (!vertexList.itemExistAt(k))
			continue;
		if (Math.abs(((Vertex)vertexList.itemAt(k)).x - x) < distance
		&& Math.abs(((Vertex)vertexList.itemAt(k)).y - y) < distance)
			return k;
	} // end for
	return NUL;
} // end of closeTo(int x, int y,int distance)


/**
*Delete the specified vertex
*@param v  the vertex to be delete.
* @exception NoSuchVertexException
*if the v is not valid.
*/
public void deleteVertex(Vertex v) throws NoSuchVertexException
{
	if (v==null)
	{
		throw new NoSuchVertexException();
	}
	deleteNode(v.getID());
}
```

```
/**
*Delete the specified node
*@param node  the node to be delete.
*@exception NoSuchVertexException
*if the node is not valid.
*/
public void deleteNode(int node) throws NoSuchVertexException
{
	Edge edge;
	if (node>=vertexSerialNo && node<0)
		throw new NoSuchVertexException("Vertex("+node+")");
	if (!vertexList.itemExistAt(node))
		throw new NoSuchVertexException("Vertex("+node+")");
	modified=true;
	for(int i=0;i<2;i++)
	{
		for(Enumeration e=edgeList.elements(node,i);e.hasMoreElements();)
		{
			edge=(Edge) e.nextElement();
			deleteEdge(edge);
		}
	}
	vertexList.deleteItemAt(node);
} // end of deleteNode(int node)
/**
*Get the vertex by node (the identity of vertex)
*@param node  the identity of vertex
*@exception NoSuchVertexException
*if the no vertex exist at index  node
*/
public Vertex getVertex(int node) throws NoSuchVertexException
{
	Vertex v=((Vertex) vertexList.itemAt(node));
	if (v==null)
		throw new NoSuchVertexException("No such Vertex["+node+"]");
	return v;
}
/**
*Purge the deleted vertex.
*Renumbumering the vertex from 0 to number of vertices
*/
public void purgeDeletedVertex()
{
```

```
        int i,j;
        int indexTable[];
        Edge edge;
        ListItem li;
        indexTable=vertexList.purgeIndexTable();
        vertexSerialNo=vertexList.count();
        if (indexTable==null)  // No vertex deleted
                return;
        modified=true;
        vertexList.purge(indexTable);
        edgeList.purge(indexTable);
        for(i=0;i<vertexSerialNo;i++)
        {
                Vertex v=(Vertex) vertexList.itemAt(i);
                v.setID(i);
                if (v.getPredecessorNode()!=NUL)
                {
                        v.setPredecessorNode(indexTable[v.getPredecessorNode()]);
                } //end if
                for(li=edgeList.getFirst(i,1);li!=null;li=edgeList.getNext(li.getCoordArray(),0))
                {
                        int coord[]={li.getCoord(0),li.getCoord(1)};
                        ((Edge) edgeList.itemAt(coord)).setVertices(coord[1],coord[0]);
                } //end for
        } // end for
} // end of purgeDeletedVertex()
/**
*Get the predecessor vertex of the specified node.
*@param node  the specified node.
*@exception NoSuchVertexException
*if the no vertex exist at index  node .
*@return the predecessor vertex of the specified node.
*/
public Vertex getPredecessor(int node) throws NoSuchVertexException
{
        Vertex v=getVertex(node);
        int pred=v.getPredecessorNode();
        if (pred!=NUL)
                return (Vertex) vertexList.itemAt(pred);
return null;
}
/**
*Get the predecessor vertex of the specified vertex.
```

```
*@param v  the specified vertex.
*@exception NoSuchVertexException
*if the v is null.
*@return the predecessor vertex of the specified vertex.
*/
public Vertex getPredecessor(Vertex v) throws NoSuchVertexException
{
        if (v==null)
                throw new NoSuchVertexException();
        int pred=v.getPredecessorNode();
        if (pred!=NUL)
                return (Vertex) vertexList.itemAt(pred);
        return null;
}
/**
*Get the predecessor node of the specified node.
*@param node  the specified node.
*@exception NoSuchVertexException
*if the no vertex exist at index  node .
*@return the predecessor node of the specified node.
*/
public int getPredecessorNode(int node) throws NoSuchVertexException
{
        Vertex v=getVertex(node);
        return v.getPredecessorNode();
}
/**
*Get the predecessor node of the specified vertex.
*@param v  the specified vertex.
*@exception NoSuchVertexException
*if the v is null.
*@return the predecessor node of the specified vertex.
*/
public int getPredecessorNode(Vertex v) throws NoSuchVertexException
{
        if (v==null)
                throw new NoSuchVertexException();
        return v.getPredecessorNode();
}
/**
*Set the predecessor vertex of the specified node.
*@param node  the node.
*@param predecessor the predecessor of <code> node </code>.
```

```
        *@exception NoSuchVertexException
        *either code or  predecessor  is not valid.
        */
        public void setPredecessor(int node,Vertex predecessor) throws NoSuchVertexException
        {
                Vertex v=getVertex(node);
                v.setPredecessorNode(predecessor.getID());
                modified=true;
        }
        /**
        *Set the predecessor vertex of the specified vertex.
        *@param v   the vertex.
        *@param predecessor the predecessor of v .
        *@exception NoSuchVertexException
        *either v or predecessor  is not valid.
        */
        public void setPredecessor(Vertex v,Vertex predecessor) throws NoSuchVertexException
        {
                if (v==null || predecessor==null)
                throw new NoSuchVertexException();
                v.setPredecessorNode(predecessor.getID());
                modified=true;
        }
/**
*Set the predecessor node of the specified node.
*@param node   the node.
*@param predecessorNode  the predecessor of <code> node </code>.
*@exception NoSuchVertexException
*either  node or predecessorNode  is not valid.
*/
        public void setPredecessor(int node,int predecessorNode) throws NoSuchVertexException
        {
                Vertex v=getVertex(node);
                v.setPredecessorNode(predecessorNode);
                modified=true;
        }
        /**
        *Set the predecessor node of the specified vertex.
        *@param v  the vertex.
        *@param predecessorNode the predecessor of  vertex .
        *@exception NoSuchVertexException
        *either  v or predecessorNode  is not valid.
        */
```

```
public void setPredecessor(Vertex v,int predecessorNode) throws NoSuchVertexException
{
	if (v==null)
	throw new NoSuchVertexException();
	v.setPredecessorNode(predecessorNode);
	modified=true;
}
/**
*Return whether the vertex exist at the index (node)
*@param node  the index (node).
*@return whether the vertex exist at the index.
*/
public boolean isVertexExist(int node)
{
	return vertexList.itemExistAt(node);
}
/**
*Return the actual number of vertices in this graph.
*Deleted vertices don't count in this method. Use getVertexSerialNo
*instead while using for-loop or while loop
*@see getVertexSerialNo
*@return the number of vertices in this graph.
*/
public final int getNumOfVertices()
{
	return vertexList.count();
}
/**
*Return the next serial number of vertices, as well as the total number of vertices, including *deleted vertices.
*Use getNumOfVertices if you don't want to count deleted vertices.
*@see getNumOfVertices
*@return the next serial number of vertices.
*/
public int getVertexSerialNo()
{
	return vertexSerialNo;
}
/**
*Returns an enumeration of the vertices of this graph.
*@returns an enumeration of the vertices of this graph.
*/
public final synchronized Enumeration verticesElements()
{
```

```
        return new VerticesEnumerator(vertexList);
    }
    /**
    *Add a clone of specified edge into this graph.
    *The method use the infomation of specified edge to create another edge.
    *@param edge the edge to add.
    */
    public void addEdge(Edge edge)
    {
        addEdge(edge.getFromNode(),edge.getToNode(),edge.weight,edge.flow,edge.color);
    }
/**
    *Add a clone of specified edge into this graph.
    *The method use the id information to add a new edge .
    *@param edge the edge to add.
    *@param id the ID of edge to add.
    */
    public void addEdge(Edge edge, int id)
    {
        addEdge(edge.getFromNode(),edge.getToNode(),edge.weight,edge.flow,edge.color);
        this.getEdge(edge.getFromNode(),edge.getToNode()).setEdgeID(id);
    }
    /**
    *Add an edge into this graph.
    *@param from  the start node of the edge.
    *@param to  the end node of the edge.
    *@param weight  the weight of the edge.
    *@param flow  the flow of the edge.
    */
    public void addEdge(int from,int to,double weight,double flow)
    {
        addEdge(from,to,weight,flow,Edge.normalColor);
    }
    /**
    *Add an edge into this graph.
    *@param from  the start node of the edge.
    *@param to  the end node of the edge.
    *@param weight  the weight of the edge.
    *@param flow  the flow of the edge.
    *@param color  the color of the edge.
    */
    public void addEdge(int from,int to,double weight,double flow, Color color)
    {
```

```
        modified=true;
        if (!directed)
        {
            if (isEdge(from,to))
            {
                return; // Already have that edge
            }else
            {
                Edge edge=new Edge(from,to,weight,flow);
                edgeList.addItemAt(edge,new int[]{to,from});
                edgeList.addItemAt(edge,new int[]{from,to},true);
                ((Vertex) vertexList.itemAt(from)).increaseOutDegree(1);
                ((Vertex) vertexList.itemAt(to)).increaseInDegree(1);
                ((Vertex) vertexList.itemAt(to)).increaseOutDegree(1);
                ((Vertex) vertexList.itemAt(from)).increaseInDegree(1);
            }
        } else
        {
            // Direct Graphs
            if (isEdge(from,to))
            {
                return; // Already have that edge
            }
            edgeList.addItemAt(new Edge(from,to,weight,flow),new int[]{to,from});
            ((Vertex) vertexList.itemAt(from)).increaseOutDegree(1);
            ((Vertex) vertexList.itemAt(to)).increaseInDegree(1);
        } // end else
        setEdge(from,to,weight,flow,color);
    } //end of addEdge(int from,int to,double weight,double flow, Color color)
    /**
    *Enumerate all edges of this graph.
    *@return graph with enumerated edges
    *@Author Baris AKTOP
    *March 2003
    */
    public Graph enumerateEdges()
    {
        int i = 0;
        Edge edge = null;
        Graph graph = this;
        for(Enumeration e=graph.allEdgesElements();e.hasMoreElements();)
        {
            edge=(Edge) e.nextElement();
```

```
                edge.setEdgeID(i);
                i++;
            }
            return graph;
    }
    /**
    *Delete an edge of this graph.
    *@param from  the start node of the edge.
    *@param to  the end node of the edge.
    */
    public void deleteEdge(int from, int to)
    {
        modified=true;
        if (!directed)
        {
            if (isEdge(from,to))
            {
                edgeList.deleteItemAt(new int[]{to,from});
                ((Vertex) vertexList.itemAt(from)).increaseOutDegree(-1);
                ((Vertex) vertexList.itemAt(to)).increaseInDegree(-1);
            } // end if
            if (isEdge(to,from))
            {
                edgeList.deleteItemAt(new int[]{from,to});
                ((Vertex) vertexList.itemAt(to)).increaseOutDegree(-1);
                ((Vertex) vertexList.itemAt(from)).increaseInDegree(-1);
            } // end if
        } else
        {
            // Directed Graph
            if (!isEdge(from,to))
            return;  // Don't have that edge
            edgeList.deleteItemAt(new int[]{to,from});
            ((Vertex) vertexList.itemAt(from)).increaseOutDegree(-1);
            ((Vertex) vertexList.itemAt(to)).increaseInDegree(-1);
        } // end if-else
    } // end of deleteEdge(int from, int to)
    /**
    *Delete an edge of this graph.
    *@param edge the edge to be delete.
    */
    public void deleteEdge(Edge edge)
    {
```

```
        int fromNode;
        int toNode;
        fromNode=edge.getFromNode();
        toNode=edge.getToNode();
        deleteEdge(fromNode,toNode);
}
/**
*Clean all special attribute of Vertices.
*@see clean
*@see cleanAllEdges
*/
public void cleanAllVertices()
{
        modified=true;
        for(Enumeration ev=vertexList.elements();ev.hasMoreElements();)
        {
                ((Vertex) ev.nextElement()).clean();
        }
}
/**
*Clean  all special attribute of Edge.
*@see clean
*@see cleanAllVertices
*/
public void cleanAllEdges()
{
        modified=true;
        for(Enumeration ea=allEdgesElements();ea.hasMoreElements();)
        {
                ((Edge) ea.nextElement()).clean();
        }
}
/**
*Delete all special edges.
*Delete edges which color is not same as  color .
*For example: If you want to keep the edge with color with Edge.normalColor,
*but delete other, you may use deleteAllOtherEdges(Edge.normalColor)
*to do this.
*@param color Ths edge with this <code> status </code> to keep.
*/
public void deleteAllOtherEdges(Color color)
{
        Edge edge;
```

```
        modified=true;
        for(Enumeration e=allEdgesElements();e.hasMoreElements();)
        {
                edge=(Edge) e.nextElement();
                if (!edge.getColor().equals(color))
                {
                        deleteEdge(edge);
                } //end if
        } //end for
} // end of deleteAllOtherEdges(Color color)

// Delete all edges
public void deleteAllEdges()
{
        modified=true;
        for(Enumeration e=allEdgesElements();e.hasMoreElements();)
        {
                deleteEdge((Edge) e.nextElement());
        }
} // end of deleteAllEdges()
/**
*Return whether the edge exist specified indexes.
*@param from  the start node of the edge.
*@param to  the end node of the edge.
*@return true if edge(from,to)  exist;  false  otherwise.
*/
public boolean isEdge(int from,int to)
{
        return (edgeList.itemExistAt(new int[]{to,from}));
}
/**
*Return whether the edge exist specified indexes. Direction don't care.
*@param from  the start node of the edge.
*@param to  the end node of the edge.
*@return true if edge(from,to) or edge(to,from)  exist;  false  otherwise.
*/
public boolean isEdgeUndir(int from,int to)
{
        return (isEdge(from,to)|| isEdge(to,from));
}


Edge getTrueEdge(Edge edge)
```

```
{
        return getTrueEdge(edge.getFromNode(),edge.getToNode());
}

Edge getTrueEdge(int from,int to)
{
        if (!isEdge(from,to))
        {
                throw new NoSuchEdgeException("No such Edge["+from+","+to+"]");
        }
        Edge edge=(Edge) edgeList.itemAt(new int[]{to,from});
        return edge;
}
/**
*Get the edge by both ends.
*@param from  the start node of the edge.
*@param to  the end node of the edge.
*@exception NoSuchEdgeException
*if the no edge exist at (from,to)
*/
public Edge getEdge(int from,int to) throws NoSuchEdgeException
{
        Edge edge=getTrueEdge(from,to);
        if (!directed)
        {
                if (edge.getToNode()==from)
                {
                        edge.reverseDirection();
                }
        }
        return edge;
} // end of getEdge(int from,int to)

public Edge getEdge(int id)
{
        Edge edge = null;
        for(Enumeration e=this.allEdgesElements();e.hasMoreElements();)
        {
                edge = (Edge) e.nextElement();
                if (edge.getEdgeID() == id)
                        break;
        }
        return edge;
```

```
} // end of getEdge(int id)
/**
*Set edge attribute.
*@param from   the start node of the edge.
*@param to    the end node of the edge.
*@param edgeWeight   the weight of the edge.
*@param edgeFlow   the flow of the edge.
*/
public void setEdge(int from,int to,double edgeWeight,double edgeFlow)
{
      setEdge(from,to,edgeWeight,edgeFlow,Edge.normalColor);
}
/**
*Set edge attribute.
*@param from   the start node of the edge.
*@param to   the end node of the edge.
*@param edgeWeight   the weight of the edge.
*@param edgeFlow   the flow of the edge.
*@param status   the status of the edge.
*@param color  the color of the edge.
*/
public void setEdge(int from,int to,double edgeWeight,double edgeFlow,Color color)
{
      modified=true;
      getEdge(from,to).setWeight(edgeWeight);
      getEdge(from,to).setFlow(edgeFlow);
      getEdge(from,to).setColor(color);
}


/**
*Return the number of edges in the graph.
*/
public int getNumOfEdges()
{
      return edgeList.count();
}
/**
*Returns an enumeration of the outgoing adjacency edges of the node of this graph.
*@param node   the node.
*@Returns an enumeration of the outgoing adjacency edges of the node of this graph.
*/
public final synchronized Enumeration edgesElements(int node)
```

```
        {
            return edgesElements(node,Graph.OUT_ADJACENCY);
        }
        /**
        *Returns an enumeration of the outgoing adjacency edges of the vertex of this graph.
        *@param v  the vertex.
        *@Returns an enumeration of the outgoing adjacency edges of the node of this graph.
        */
        public final synchronized Enumeration edgesElements(Vertex v)
        {
            return edgesElements(v.getID(),Graph.OUT_ADJACENCY);
        }
        /**
        *Returns an enumeration of the adjacency edges of the node of this graph.
        *@param node  the node.
        *@param dimension  Graph.OUT_ADJACENCY : return the enumeration
        *of outgoing adjacency edges; Graph.IN_ADJACENCY return
        *the enumeration of incoming adjacency edges.
        *@Returns an enumeration of the adjacency edges of the node of this graph.
        */
        public final synchronized Enumeration edgesElements(int node,int dimension)
        {
            return new EdgesEnumerator(edgeList,node,dimension,directed);
        }
        /**
        *Returns an enumeration of the adjacency edges of the vertex of this graph.
        *@param v  the vertex.
        *@param dimension Graph.OUT_ADJACENCY  : return the enumeration of outgoing *adjacency edges;
        Graph.IN_ADJACENCY returnthe enumeration of incoming adjacency *edges.
        *@Returns an enumeration of the adjacency edges of the node of this graph.
        */
        public final synchronized Enumeration edgesElements(Vertex v,int dimension)
        {
            return new EdgesEnumerator(edgeList,v.getID(),dimension,directed);
        }
    /**
    *Returns an enumeration of the adjacency vertex of the node of this graph.
        *@param node  the node.
        *@Returns an enumeration of the adjacency vertex of the node of this graph.
        *Directed graph is treat as undirected graph.
        */
        public final synchronized Enumeration directedAdjVerticesElements(int node)
        {
```

```
        return new DirectedAdjVerticesEnumerator(this,node,Graph.OUT_ADJACENCY);
}
/**
*Returns an enumeration of the adjacency vertex of the node of this graph.
*@param v   the vertex.
*@param dimension   Graph.OUT_ADJACENCY : return the enumeration of outgoing *adjacency edges; Graph.IN_ADJACENCY return the enumeration of incoming adjacency *edges.
*@Returns an enumeration of the adjacency vertex of the node of this graph.
*Directed graph is treat as undirected graph.
*/
public final synchronized Enumeration directedAdjVerticesElements(int node,int dimension)
{
        return new DirectedAdjVerticesEnumerator(this,node,dimension);
}
/**
*Returns an enumeration of the adjacency vertex of  v of this graph.
*@param v   the Vertex.
*@Returns an enumeration of the adjacency vertex of the node of this graph.
*Directed graph is treat as undirected graph.
*/
public final synchronized Enumeration directedAdjVerticesElements(Vertex v)
{
        return new DirectedAdjVerticesEnumerator(this,v.getID(),Graph.OUT_ADJACENCY);
}
/**
*Returns an enumeration of the adjacency vertex of  v  of this graph.
*@param v   the Vertex.
*@param dimension Graph.OUT_ADJACENCY : return the enumeration
*of outgoing adjacency edges; Graph.IN_ADJACENCY return
*the enumeration of incoming adjacency edges.
*@Returns an enumeration of the adjacency vertex of the node of this graph.
*Directed graph is treat as undirected graph.
*/
public final synchronized Enumeration directedAdjVerticesElements(Vertex v,int dimension)
{
        return new DirectedAdjVerticesEnumerator(this,v.getID(),dimension);
}
/**
*Returns an enumeration of the adjacency vertex of the node of this graph.
*Directed graph is treat as undirected graph.
*@param node  the node.
*@Returns an enumeration of the adjacency vertex of the node of this graph.
*Directed graph is treat as undirected graph.
```

```
        */
        public final synchronized Enumeration undirectedAdjVerticesElements(int node)
        {
                return new UndirectedAdjVerticesEnumerator(this,node);
        }
        /**
        *Returns an enumeration of the adjacency vertex of  v of this graph.
        *Directed graph is treat as undirected graph.
        *@param Vertex  the vertex.
        *@Returns an enumeration of the adjacency vertex of the node of this graph.
        *Directed graph is treat as undirected graph.
        */
        public final synchronized Enumeration undirectedAdjVerticesElements(Vertex v)
        {
                return new UndirectedAdjVerticesEnumerator(this,v.getID());
        }
        /**
        *Returns an enumeration of the all edges of this graph.
        *@returns an enumeration of the all edges of this graph.
        */
        public final synchronized Enumeration allEdgesElements()
        {
                return new AllEdgesEnumerator(this);
        }
} // end of class Graph

class VerticesEnumerator implements Enumeration
{
        protected ListArray la;
        protected Enumeration e;
        protected VerticesEnumerator(ListArray a)
        {
                la=a;
                e=la.elements();
        }

        public boolean hasMoreElements()
        {
                return e.hasMoreElements();
        }

        public Object nextElement()
        {
```

```
            synchronized (la)
            {
                if (e.hasMoreElements())
                {
                    return e.nextElement();
                }
            }
            throw new NoSuchElementException("VerticesEnumerator");
        }
    } // end of class VerticesEnumerator

    class EdgesEnumerator implements Enumeration
    {
        protected ListArray2 la2;
        protected Enumeration e;
        protected boolean directed;
        protected int fromNode;
        protected EdgesEnumerator(ListArray2 la2,int node,int dimension,boolean directed)
        {
            this.la2=la2;
            e=la2.elements(node,dimension);
            this.directed=directed;
            fromNode=node;
        }
        public boolean hasMoreElements()
        {
            return e.hasMoreElements();
        }
        public Object nextElement()
        {
            synchronized (la2)
            {
                if (e.hasMoreElements())
                {
                    Edge edge=(Edge) e.nextElement();
                    if (!directed)
                    {
                        if (edge.getToNode()==fromNode)
                        {
                            edge.reverseDirection();
                        }
                    }
                    return (Object) edge;
```

```
                }
            }
            throw new NoSuchElementException("EdgeEnumerator");
        }
} // end of class EdgesEnumerator

class DirectedAdjVerticesEnumerator implements Enumeration
{
    protected Graph graph;
    protected Enumeration e;
    protected boolean directed;
    protected int node;
    protected Edge edge;
    protected int dimension;
    protected DirectedAdjVerticesEnumerator(Graph graph,int node,int dimension)
    {
        this.graph=graph;
        e=graph.edgesElements(node,dimension);
        this.directed=graph.directed;
        this.node=node;
        this.dimension=dimension;
    }
    public boolean hasMoreElements()
    {
        if (e==null)
            return false;
        return (e.hasMoreElements());
    }
    public Object nextElement()
    {
        Vertex v;
        synchronized (graph)
        {
            if (e.hasMoreElements())
            {
                edge=(Edge) e.nextElement();

                if (dimension==Graph.IN_ADJACENCY)
                {
                    v=graph.getVertex(edge.getFromNode());
                } else
                {
                    v=graph.getVertex(edge.getToNode());
```

```
                    }
                    return (Object) v;
                }
            }
            throw new NoSuchElementException("DirectedAdjVerticesEnumerator");
        }
} //end of class DirectedAdjVerticesEnumerator

class UndirectedAdjVerticesEnumerator implements Enumeration
{
        protected Graph graph;
        protected Enumeration e;
        protected boolean directed;
        protected int node;
        protected Edge edge;
        protected int dimension;

        protected UndirectedAdjVerticesEnumerator(Graph graph,int node)
        {
            this.graph=graph;
            e=graph.edgesElements(node,Graph.OUT_ADJACENCY);
            this.directed=graph.directed;
            this.node=node;
            findNextElement();
            dimension=Graph.OUT_ADJACENCY;
        }
        private synchronized void findNextElement()
        {
            do{
                    while(e.hasMoreElements())
                    {
                        edge=(Edge) e.nextElement();
                        if ((edge.getFromNode()==edge.getToNode()) && (dimension == graph.IN_ADJACENCY) )
                        {
                            continue;
                        }
                        if (!e.hasMoreElements())
                        {
                            break;
                        }
                        return;
                    } //end while
                    if (dimension==Graph.OUT_ADJACENCY && directed)
```

```
                {
                    dimension=Graph.IN_ADJACENCY;
                    e=graph.edgesElements(node,dimension);
                }
            }while(e.hasMoreElements());
} //end of findNextElement()
public boolean hasMoreElements()
{
    if (e==null)
        return false;
    return (edge!=null) ;
}
public Object nextElement()
{
    Vertex v;
    synchronized (graph)
    {
        if (e.hasMoreElements() )
        {
            if (directed && edge.getToNode()==node)
            {
                v=graph.getVertex(edge.getFromNode());
            } else
            {
                v=graph.getVertex(edge.getToNode());
            }
            findNextElement();
            return (Object) v;
        }else if (edge!=null)
        {
            if (directed && edge.getToNode()==node)
            {
                v=graph.getVertex(edge.getFromNode());
            } else
            {
                v=graph.getVertex(edge.getToNode());
            }
            edge=null;
            return (Object) v;
        }
    }
    throw new NoSuchElementException("UndirectedAdjVerticesEnumerator");
}
```

```
} // end of class UndirectedAdjVerticesEnumerator

class AllEdgesEnumerator implements Enumeration
{
	protected ListArray2 la2;
	protected Enumeration e=null;
	protected Vertex v;
	protected Edge edge=null;
	protected Graph graph;

	AllEdgesEnumerator(Graph graph)
	{
		this.graph=graph;
		try
		{
			e=graph.edgeList.allElements();
		}catch(NoSuchElementException ne)
		{
			return;
		}
	} //end of  AllEdgesEnumerator
	public boolean hasMoreElements()
	{
		return (e!=null && e.hasMoreElements());
	}
	public Object nextElement()
	{
		synchronized (graph)
		{
			if (hasMoreElements())
			{
				return (Object) e.nextElement();
			}
		}
		throw new NoSuchElementException("AllEdgeEnumerator");
	}
} // end of class AllEdgesEnumerator
```

5. Edge.java

```
package tw.edu.ncnu.im.cnclab.JGAP;
import tw.edu.ncnu.im.cnclab.DataStru.*;
import java.io.Serializable;
```

```
import java.awt.Color;
/**
*The Edge class implements data structure of edge of graphs
*@see java.util.Enumeration
*@see Vertex
*@see Graph
*/
public class Edge implements Cloneable, Comparable11 , Serializable
{
	//Indicate null item
	private int pathID = 0;
	public static final int NUL = -1;
	static final long serialVersionUID = -5946460600638491899L;
	public boolean critical = false;
	//Indicate node that edge begins with.
	protected int fromNode;
	//Indicate node that edge ends with.
	protected int toNode;
	//Weight of the Edge.
	protected double weight;
	//Flow of the Edge.
	protected double flow;
	//Default color of the normal Edge.
	public static Color normalColor=Color.black;
	//Default color  of the untouched Edge.
	public static Color untouchedColor=Color.magenta;
	//Status of the Edge. Use color-representation.
	protected Color color;
	//Id of edge
	int edgeSerialNo = 0;
	//Constructs a dummy edge.
	public Edge()
	{
		this(NUL,NUL,0.0,0.0,normalColor);
	}
	/**
	*Constructs an edge with specified data.
	*@param fromNode   Node that edge begin with.
	*@param toNode   Node that edge end with.
	*@param weight   Edge weight
	*@param flow  Initial flow
	*/
	public Edge(int fromNode,int toNode,double weight, double flow)
```

```
{
        this(fromNode,toNode,weight,flow,normalColor);
}
/**
*Constructs an edge with specified data and color.
*@param fromNode   Node that edge begin with.
*@param toNode   Node that edge end with.
*@param weight   Edge weight
*@param flow   Initial flow
*@param color   Color of edge
*/
public Edge(int fromNode,int toNode,double weight, double flow,Color color)
{
        this.toNode=toNode;
        this.fromNode=fromNode;
        this.weight=weight;
        this.flow=flow;
        this.color=color;
}
/**
*Clean all special attribute.
*It will call reset() and reset the following attribute: color=normalColor; flow=0.0;
*@see reset
*/
public void clean()
{
        reset();
        flow=0.0;
}
public void setEdgeID(int id)
{
        this.edgeSerialNo =id;
}
public int getEdgeID()
{
        return this.edgeSerialNo;
}
public void setPathID(int id)
{
        this.pathID = id;
}
public int getPathID ()
{
```

```
        return this.pathID;
}
/**
*Reset color attribute.
*Reset the following attribute: color=normalColor;
*@see clear
*/
public void reset()
{
        color=normalColor;
}
/**
*Returns a deep-copied clone of this Edge.
*@return a deep-copied clone of this Edge.
*/
public Object clone()
{
        try
        {
                Edge e=(Edge)super.clone();
                e.copyFrom(this);
                return e;
        }catch (CloneNotSupportedException e)
        {
                throw new InternalError();
        }
}
/**
*Compares edges with their weight.
*@param rhs the other Edge object.
*@return 0 if two objects are equal;
*-1 if this object is smaller;
*1 if this object is larger.
*@exception ClassCastException if rhs is no a Edge.
*/
public int compareTo(Object rhs) throws ClassCastException
{
        if (weight<((Edge) rhs).weight)
                return -1;
        if (weight>((Edge) rhs).weight)
                return 1;
        return 0;
}
```

```
/**
*Copies the components from specified Edge to this Edge
*@param  src the source Edge
*/
public void copyFrom(Edge src)
{
        fromNode=src.fromNode;
        toNode=src.toNode;
        weight=src.weight;
        flow=src.flow;
        color=src.color;
}
/**
*Get the color of edge.
*@return the color of edge.
*/
public Color getColor()
{
        return color;
}
/**
*Set the color of edge.
*@param color  the color of edge.
*/
public void setColor(Color color)
{
        this.color=color;
}

/**
*Get the flow of edge.
*@return the flow of edge.
*/
public double getFlow()
{
        return flow;
}
/**
*Set the flow of edge.
*@param flow   the flow of edge.
*/
public void setFlow(double flow)
{
```

```
        this.flow=flow;
}
/**
*Get the default color of edge.
*@return the default color of edge.
*/
public Color getDefaultColor()
{
        return normalColor;
}
/**
*Set the default color of edge.
*@param color  the default color of edge.
*/
public void setDefaultColor(Color color)
{
        this.normalColor=color;
}
/**
*Get the start node of edge.
*@return the start node of edge.
*/
public int getFromNode()
{
        return fromNode;
}
/**
*Set the start node of edge.
*@param fromNode  the start node of edge.
*/
public void setFromNode(int fromNode)
{
        this.fromNode=fromNode;
}
/**
*Get the end node of edge.
*@return the end node of edge.
*/
public int getToNode()
{
        return toNode;
}
/**
```

```
*Set the end node of edge.
*@param toNode  the end node of edge.
*/
public void setToNode(int toNode)
{
	this.toNode=toNode;
}
/**
*Set the both end of edge
*@param fromNode  the start node of edge
*@param toNode    the end node of edge
*/
public void setVertices(int fromNode,int toNode)
{
	this.toNode=toNode;
	this.fromNode=fromNode;
}
/**
*Reverse the direction of this edge.
*/
public void reverseDirection()
{
	setVertices(toNode,fromNode);
}
/**
*Get the weight of edge.
*@return the weight of edge.
*/
public double getWeight()
{
	return weight;
}
/**
*Set the weight of edge.
*@param weight   the weight of edge.
*/
public void setWeight(double weight)
{
	this.weight=weight;
}
//return if this edge is critical
public boolean getCritical()
{
```

```
            return this.critical;
        }
        //set this edge as critical
        public void setCritical(boolean p)
        {
            this.critical = p;
        }
        //flow is equal to ke
        public int getKe ()
        {
            double flow = this.getFlow();
            int Ke = (int)flow;
            return Ke;
        }
} // end of class Edge
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[AKT02] Baris Aktop, Ekrem Serin, Selcuk Ozturk, *CS4552 Network Design and Programming: Project#3,* NPS Class Project, June 2002

[CAR79] Carey M., and Johnson, D., *Computers and Intractability: A Guide to the Theory of NP-Completeness,* WH Freeman, San Francisco, 1979

[DIN01] Ding-Yi Chen, Tyng-Ruey, and Shi-Chun Tsai, "*JGAP: a Java-based graph algoritms platform"*, Sotware-practice and experience, 2001;31:615-635

[ELL97] Robert J. Ellison, David A. Fisher, Richard C. Linger, Howard F. Lipson, Thomas A. Longstaff, Nancy R. Mead, *Survivable Network Systems: An Emerging Discipline,* CERT Coordination Center Software Engineering Institute Carnegie Melon University, November 1997

[ELL99] Robert J. Ellison, David A. Fisher, Richard C. Linger, Howard F. Lipson, Thomas A. Longstaff, Nancy R. Mead, *Survivability: Protecting Your Critical Systems,* CERT Coordination Center Software Engineering Institute Carnegie Melon University, 1999

[FIS99] David A. Fisher, Howard F. Lipson, *Emergent Algorithms: A New Method for Enhancing Survivability in Unbounded Systems,* Software Engineering Institute Carnegie Melon University, 1999

[GIB85] Alan Gibbons, *Algorithmic Graph Theory,* pp. 98-111, Cambridge University Press, 1985

[JHA00] Jha, S. and Wing, J. M. , "*Survivability Analysis of A Networked System"*, Proceedings of the 23rd International Conference of Software Engineering, pp. 307-317, July 2001

[KAT00] Efraim Kati, *Fault-Tolerant Approach for Deploying Server Agent-based Active network Management (SAAM) in Windows NT Environment to Provide Uninterrupted Services to Routers in case of Server Failure(s),* Master's Thesis Naval Post Graduate School, Monterey, California, March 2000

[LIN99] Richard C. Linger, Robert J. Ellison, Thomas A. Longstaff, Nancy R. Mead, *The Survivability Imperative: Protecting Critical Systems,* Software Engineering Institute Carnegie Melon University, 1999

[MAR01] Scott Margulis, *MAGMA: A liquid Software Approach to Fault Tolerance Computer Network Security and Survivable Networking*, Master's Thesis Naval Post Graduate School, Monterey, California, December 2001

[MEA99] R. C. Linger, N. R. Mead, H. F. Lipson, *Requirements Definition for Survivable Network Systems,* Software Engineering Institute Carnegie Melon University, 1999

[MEA00] Mead N.R. *et al.* , "Survivable Network Analysis Method", Technical report, Carnegie Mellon Software Engineering Institute, CMU/SEI-2000-TR-013, September 2000

[SUL99] Sullivan, K. *et al.* , "Information Survivability Control Systems", *Proceedings of 1999 International Conference of Software Engineering* (ICSE 99), pp.184-192, Los Alamitos, May 1999

[UMA01] Umar, A. *et al.* , "Intrusion Tolerant Middleware", *Proceedings of DARPA Information Survivability Conference and Exposition* (DISCEX 01) vol. 2, 2001

[WEL00] Wells, D. *et al.* ,"Software Survivability", *Proceedings of DARPA Information Survivability Conference and Exposition* (DISCEX 01) vol. 2, 2000

[XIE98] Geoffrey G. Xie, "SAAM: An Integrated Network Architecture for Integrated Services", paper presented at the 6th IEEE/IFIP International Workshop on Quality of Service, NAPA, California, May 1998

[XIE02] Geoffrey G. Xie, CY03 Homeland Security Research & Technology Research Proposal: "Critical Infrastructure Protection: Maximize Survivability of a Network Dependent Service," 2002

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Geoffrey Xie
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

4. Mr. John Gibson
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

5. Deniz Kuvvetleri Komutanligi
   Kutuphane
   Bakanliklar, Ankara, TURKEY

6. Deniz Harp Okulu Komutanligi
   Kutuphane
   Tuzla, Istanbul, TURKEY

7. Bilkent Universitesi Kutuphanesi
   Bilkent, Ankara, TURKEY

8. Orta Dogu Teknik Universitesi Kutuphanesi
   Balgat, Ankara, TURKEY

9. Bogazici Universitesi Kutuphanesi
   Bebek, Istanbul, TURKEY

10. Dz. Utgm. Baris Aktop
    Envanter Kontrol Merkezi Komutanligi
    Golcuk, Kocaeli, TURKEY